

# ***Server-Side Communication ActionScript Dictionary***

---

***Macromedia Flash™ Communication Server MX 1.5***



## Trademarks

Afterburner, AppletAce, Attain, Attain Enterprise Learning System, Attain Essentials, Attain Objects for Dreamweaver, Authorware, Authorware Attain, Authorware Interactive Studio, Authorware Star, Authorware Synergy, Backstage, Backstage Designer, Backstage Desktop Studio, Backstage Enterprise Studio, Backstage Internet Studio, Contribute, Design in Motion, Director, Director Multimedia Studio, Doc Around the Clock, Dreamweaver, Dreamweaver Attain, Drumbeat, Drumbeat 2000, Extreme 3D, Fireworks, Flash, Fontographer, FreeHand, FreeHand Graphics Studio, Generator, Generator Developer's Studio, Generator Dynamic Graphics Server, Knowledge Objects, Knowledge Stream, Knowledge Track, Lingo, Live Effects, Macromedia, Macromedia M Logo & Design, Macromedia Contribute, Macromedia Flash, Macromedia Xres, Macromind, Macromind Action, MAGIC, Mediamaker, Object Authoring, Power Applets, Priority Access, Roundtrip HTML, Scriptlets, SoundEdit, ShockRave, Shockmachine, Shockwave, Shockwave Remote, Shockwave Internet Studio, Showcase, Tools to Power Your Ideas, Universal Media, Virtuoso, Web Design 101, Whirlwind and Xtra are trademarks of Macromedia, Inc. and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words or phrases mentioned within this publication may be trademarks, servicemarks, or tradenames of Macromedia, Inc. or other entities and may be registered in certain jurisdictions including internationally.

## Third-Party Information

Speech compression and decompression technology licensed from Nellymoser, Inc. ([www.nellymoser.com](http://www.nellymoser.com)).



Sorenson™ Spark™ video compression and decompression technology licensed from  
Sorenson Media, Inc.

This guide contains links to third-party websites that are not under the control of Macromedia, and Macromedia is not responsible for the content on any linked site. If you access a third-party website mentioned in this guide, then you do so at your own risk. Macromedia provides these links only as a convenience, and the inclusion of the link does not imply that Macromedia endorses or accepts any responsibility for the content on those third-party sites.

**Copyright © 2002-2003 Macromedia, Inc. All rights reserved. This manual may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without prior written approval of Macromedia, Inc.**

## Acknowledgments

Director: Erick Vera

Producer: JuLee Burdekin

Writing: Jay Armstrong, Jody Bleyle, JuLee Burdekin, Barbara Herbert, Barbara Nelson, Shimul Rahim, Tim Statler

Managing Editor: Rosana Francescato

Editing: Mary Ferguson, Anne Szabla

Production Manager: Patrice O'Neill

Multimedia Design and Production: Aaron Begley, Benjamin Salles

Print Design and Production: Chris Basmajian

Second Edition: March 2003

Macromedia, Inc.  
600 Townsend St.  
San Francisco, CA 94103

# CONTENTS

Server-Side Communication ActionScript . . . . .	5
Using server-side ActionScript . . . . .	5
Using naming conventions . . . . .	6
Contents of the dictionary. . . . .	7
Application (object). . . . .	9
Application.acceptConnection. . . . .	10
Application.clearSharedObjects . . . . .	11
Application.clearStreams . . . . .	12
Application.clients. . . . .	14
Application.disconnect . . . . .	14
Application.getStats. . . . .	15
Application.hostname . . . . .	15
Application.name . . . . .	16
Application.onAppStart. . . . .	16
Application.onAppStop . . . . .	17
Application.onConnect . . . . .	18
Application.onConnectAccept. . . . .	21
Application.onConnectReject . . . . .	22
Application.onDisconnect . . . . .	23
Application.onStatus . . . . .	24
Application.registerClass . . . . .	25
Application.registerProxy. . . . .	26
Application.rejectConnection . . . . .	27
Application.server . . . . .	28
clearInterval. . . . .	29
Client (object). . . . .	29
Client.agent. . . . .	32
Client.call . . . . .	33
Client."commandName". . . . .	34
Client.getBandwidthLimit. . . . .	35
Client.getStats. . . . .	36
Client.ip . . . . .	37
Client.protocol . . . . .	37
Client.ping . . . . .	38
Client.readAccess. . . . .	38
Client.referrer . . . . .	39
Client.__resolve. . . . .	39
Client.setBandwidthLimit. . . . .	40

Client.writeAccess . . . . .	41
load . . . . .	41
NetConnection (object) . . . . .	42
NetConnection.addHeader . . . . .	43
NetConnection.call . . . . .	44
NetConnection.close . . . . .	45
NetConnection.connect . . . . .	46
NetConnection.isConnected . . . . .	47
NetConnection.onStatus . . . . .	47
NetConnection.uri . . . . .	48
setInterval . . . . .	48
SharedObject (object) . . . . .	49
SharedObject.clear . . . . .	51
SharedObject.close . . . . .	52
SharedObject.flush . . . . .	53
SharedObject.get . . . . .	53
SharedObject.getProperty . . . . .	55
SharedObject.getPropertyNames . . . . .	56
SharedObject.handlerName . . . . .	56
SharedObject.lock . . . . .	57
SharedObject.name . . . . .	57
SharedObject.onStatus . . . . .	58
SharedObject.onSync . . . . .	58
SharedObject.purge . . . . .	60
SharedObject.resyncDepth . . . . .	61
SharedObject.send . . . . .	61
SharedObject.setProperty . . . . .	62
SharedObject.size . . . . .	63
SharedObject.unlock . . . . .	64
SharedObject.version . . . . .	64
Stream (object) . . . . .	65
Stream.bufferTime . . . . .	66
Stream.clear . . . . .	66
Stream.get . . . . .	67
Stream.length . . . . .	67
Stream.name . . . . .	68
Stream.onStatus . . . . .	68
Stream.play . . . . .	69
Stream.record . . . . .	72
Stream.send . . . . .	73
Stream.setBufferTime . . . . .	74
trace . . . . .	75

## APPENDIX

Server-Side Information Objects . . . . .	77
---	----

# Server-Side Communication ActionScript

Server-Side Communication ActionScript is a scripting language on the server that lets you develop efficient and flexible client-server Macromedia Flash Communication Server MX 1.5 applications. For example, you can use server-side ActionScript to control login procedures, control events in connected Macromedia Flash movies, determine what users see in their Flash movies, and communicate with other servers. You can also use server-side scripting to allow and disallow users access to various server-side application resources and to allow users to update and share information.

Server-side ActionScript is based on the ECMA-262 specification (ECMAScript 1.5) derived from JavaScript and lets you access the core JavaScript server object model. (For more information, see the Netscape DevEdge website at <http://developer.netscape.com/docs/manuals/index.html?content=ssjs.html>.) Server-side ActionScript provides global methods and objects and exposes a rich object model for developing communication applications. You can also create your own objects, properties, and methods. This dictionary provides detailed information about the objects and their properties, methods, and events.

Client-Side Communication ActionScript is based on the ECMA-262 specification, but there are some differences in its implementation. Server-side ActionScript, however, does not deviate from the ECMA-262 specification. For information about the relationship between server-side ActionScript and client-side ActionScript, see *Developing Communication Applications*.

## Using server-side ActionScript

To use server-side ActionScript with a Flash Communication Server application, you write the code, copy the script into the appropriate server directory, and run the SWF file that connects to the server. To understand Flash Communication Server applications, see *Developing Communication Applications*.

Create the server-side ActionScript file and name it `main.asc`. All ActionScript code that is embedded in the script file and not inside a function body executes once when the application is loaded but before the `application.onAppStart` event handler is called.

**Note:** You can also name your server-side script file `app_name`, where `app_name` is the name of your application's directory, and save it with a file extension of `.asc` or `.js`. Also, any double-byte characters (including characters of all Asian languages) in the server-side ActionScript file must be UTF-8-encoded. For more information on server-side script files, see *Developing Communication Applications*.

To install and test the server-side ActionScript file, do the following:

- 1 Locate the Macromedia Flash Communication Server /applications directory.

The default location of the /applications directory is under the Macromedia Flash Communication Server MX product installation directory.

**Note:** If you did not accept the default installation settings and you aren't sure where the application directory is located, the location is specified in the `<AppSDir>` tag of the `Vhost.xml` file, which is located at `\Flash Communication Server MX\conf\defaultRoot_\defaultVhost\_`. For information about configuring a different application directory, see *Managing Flash Communication Server*. While your SWF and HTML files should be published under a web server directory, your server-side ASC files, your audio/video FLV files, and your ActionScript FLA source files should not be accessible to a user browsing your website.

- 2 Your server-side script file must be named `main.asc`, `main.js`, `registered_app_name.asc`, or `registered_app_name.js`.
- 3 Create a subdirectory in the /applications directory called *appName*, where *appName* is a name you choose as the filename of your Flash Communication Server application. You must pass this name as a parameter to the `NetConnection.connect` method in the client-side ActionScript.
- 4 Place the `main.asc` file in the *appName* directory or in a subdirectory called `scripts` in the *appName* directory.
- 5 Open the Flash application (the SWF) in a browser or in the stand-alone Flash Player.

The SWF must contain ActionScript code that passes *appName* to the `connect` method of the `NetConnection` object, as shown in the following example:

```
nc = new NetConnection();
nc.connect("rtmp://flashcomsvr.mydomain.com/myFlashComAppName");
```

**Note:** You can use the Communication App inspector or the Administration Console to check if the application loaded successfully.

## Using naming conventions

When you write server-side ActionScript code, there are certain naming conventions that you must use to name your applications, methods, properties, and variables. These rules let you logically identify objects so your code executes properly.

### Naming applications

Flash Communication Server application names must follow the Uniform Resource Identifier (URI) RFC 2396 convention (see <http://www.w3.org/Addressing/>). This convention supports a hierarchical naming system where a forward slash (/) separates the elements in the hierarchy. The first element specifies the application name. The element following the application name specifies the application instance name. Each instance of the application has its own script environment.

### Specifying instances

By specifying a unique application instance name after an application name, you can run multiple instances of a single application. For example, `rtmp://support/session215` specifies a customer support application named “support” and refers to a specific session of that application named “session215”. All users who connect to the same instance name can communicate with each other by referencing the same streams or shared objects.

## Using JavaScript syntax

You must follow all syntax rules of JavaScript. For example, JavaScript is case-sensitive and does not allow punctuation other than underscores ( `_` ) and dollar signs ( `$` ) in names. You can use numbers in names, but names cannot begin with a number.

## Avoiding reserved commands

Flash Communication Server has reserved commands that you cannot use in a script. These commands are either methods that belong to the client-side `NetConnection` object or methods that belong to the server-side `Client` object. This means that if you have a `NetConnection` object on the client (player), you cannot make the following call:

```
nc.call( "reservedCmd", ... );
```

In this call, "reservedCmd" is any of the following commands: `closeStream`, `connect`, `createStream`, `deleteStream`, `onStatus`, `pause`, `play`, `publish`, `receiveAudio`, `receiveVideo`, or `seek`. It also cannot be any of the server-side `Client` object methods: `getBandwidthLimit`, `setBandwidthLimit`, `getStats`, and `ping`.

## Contents of the dictionary

All dictionary entries are listed alphabetically. However, methods, properties, and event handlers that are associated with an object are listed along with the object's name—for example, the `name` property of the `Application` object is listed as `Application.name`. The following table helps you locate these elements.

ActionScript element	See entry
<code>acceptConnection</code>	<code>Application.acceptConnection</code>
<code>addHeader</code>	<code>NetConnection.addHeader</code>
<code>agent</code>	<code>Client.agent</code>
<code>Application</code>	<code>Application</code> (object)
<code>bufferTime</code>	<code>Stream.bufferTime</code>
<code>call</code>	<code>Client.call</code> , <code>NetConnection.call</code>
<code>clear</code>	<code>SharedObject.clear</code> , <code>Stream.clear</code>
<code>clearInterval</code>	<code>clearInterval</code>
<code>clearSharedObject</code>	<code>Application.clearSharedObjects</code>
<code>clearStreams</code>	<code>Application.clearStreams</code>
<code>Client</code>	<code>Client</code> (object)
<code>clients</code>	<code>Application.clients</code>
<code>close</code>	<code>NetConnection.close</code> , <code>SharedObject.close</code>
<code>"commandName"</code>	<code>Client."commandName"</code>
<code>connect</code>	<code>NetConnection.connect</code>
<code>disconnect</code>	<code>Application.disconnect</code>
<code>flush</code>	<code>SharedObject.flush</code>
<code>get</code>	<code>SharedObject.get</code> , <code>Stream.get</code>

---

getBandwidthLimit	Client.getBandwidthLimit
getProperty	SharedObject.getProperty
getPropertyNames	SharedObject.getPropertyNames
handlerName	SharedObject.handlerName
hostName	Application.hostname
ip	Client.ip
isConnected	NetConnection.isConnected
length	Stream.length
load	load
lock	SharedObject.lock
name	Application.name, SharedObject.name, Stream.name
<b>NetConnection</b>	<b>NetConnection (object)</b>
onAppStart	Application.onAppStart
onAppStop	Application.onAppStop
onConnect	Application.onConnect
onConnectAccept	Application.onConnectAccept
onConnectReject	Application.onConnectReject
onDisconnect	Application.onDisconnect
onStatus	Application.onStatus, NetConnection.onStatus, SharedObject.onStatus, Stream.onStatus
onSync	SharedObject.onSync
play	Stream.play
purge	SharedObject.purge
readAccess	Client.readAccess
record	Stream.record
referrer	Client.referrer
registerClass	Application.registerClass
registerProxy	Application.registerProxy
rejectConnection	Application.rejectConnection
__resolve	Client.__resolve
resyncDepth	SharedObject.resyncDepth
send	SharedObject.send, Stream.send
setBandwidthLimit	Client.setBandwidthLimit
setBufferTime	Stream.setBufferTime
setInterval	setInterval
setProperty	SharedObject.setProperty
<b>SharedObject</b>	<b>SharedObject (object)</b>

---

---

size	SharedObject.size
server	Application.server
Stream	Stream (object)
trace	trace
unlock	SharedObject.unlock
uri	NetConnection.uri
version	SharedObject.version
writeAccess	Client.writeAccess

---

## Application (object)

The Application object contains information about a Flash Communication Server application instance that lasts until the application instance is unloaded. A Flash Communication Server application is a collection of stream objects, shared objects, and clients (connected users). Each application has a unique name, and you can use the naming scheme described in Using naming conventions to create multiple instances of an application.

The Application object lets you accept and reject client connection attempts, register and unregister classes and proxies, and create functions that are invoked when an application starts or stops or when a client connects or disconnects.

Besides the built-in properties of the Application object, you can create other properties of any legal ActionScript type, including references to other ActionScript objects. For example, the following lines of code create a new property of type `array` and a new property of type `number`:

```
application.myarray = new Array();
application.num_requests = 1;
```

## Method summary for the Application object

Method	Description
<code>Application.acceptConnection</code>	Accepts a connection to an application from a client.
<code>Application.clearSharedObjects</code>	Clears all shared objects associated with the current instance.
<code>Application.clearStreams</code>	Clears all stream objects associated with the current instance.
<code>Application.disconnect</code>	Disconnects a client from the server.
<code>Application.getStats</code>	Returns network statistics for the application instance.
<code>Application.registerClass</code>	Registers or unregisters a constructor that is called during object deserialization.
<code>Application.registerProxy</code>	Registers a <code>NetConnection</code> or <code>Client</code> object to fulfill a method request.
<code>Application.rejectConnection</code>	Rejects a connection to an application.

## Property summary for the Application object

Property (read-only)	Description
<code>Application.clients</code>	An object containing a list of all clients currently connected to the application.
<code>Application.hostname</code>	The hostname of the server for default virtual hosts, and virtual hostname for non-default virtual hosts.
<code>Application.name</code>	The name of an application instance.
<code>Application.server</code>	The platform and version of the server.

## Event handler summary for the Application object

Event handler	Description
<code>Application.onAppStart</code>	Invoked when the application is loaded by the server.
<code>Application.onAppStop</code>	Invoked when the application is unloaded by the server.
<code>Application.onConnect</code>	Invoked when a client connects to the application.
<code>Application.onConnectAccept</code>	Invoked when a client successfully connects to the application; for use with communication components only.
<code>Application.onConnectReject</code>	Invoked when a client fails to connect to the application; for use with communication components only.
<code>Application.onDisconnect</code>	Invoked when a client disconnects from the application.
<code>Application.onStatus</code>	Invoked when a script generates an error.

## Application.acceptConnection

### Availability

Flash Communication Server MX.

### Usage

```
application.acceptConnection(clientObj)
```

**Parameters**

*clientObj* A client to accept.

**Returns**

Nothing.

**Description**

Method; accepts the connection call from a client to the server. The `application.onConnect` event handler is invoked on the server side to notify a script when `NetConnection.connect` is called from the client side. You can use the `application.acceptConnection` method inside an `application.onConnect` event handler to accept a connection from a client. You can use the `application.acceptConnection` method outside an `application.onConnect` event handler to accept a client connection that had been placed in a pending state (for example, to verify a user name and password).

When you use components and your code includes an explicit call to `application.acceptConnection` or `application.rejectConnection`, the last line (in order of execution) of the `onConnect` method should be either `application.acceptConnection` or `application.rejectConnection`. Also, any logic that follows the explicit `acceptConnection` or `rejectConnection` statement must be placed in `application.onConnectAccept` and `application.onConnectReject` statements, or it will be ignored. This requirement exists only when you use components.

**Example**

The following example uses the `application.acceptConnection` method to accept the connection from `client1`:

```
application.onConnect = function (client1){
    // insert code here
    application.acceptConnection(client1);
    client1.call("welcome");
};
```

**Note:** The example above shows code from an application that does not use components.

**See also**

`Application.onConnect`, `Application.rejectConnection`

## Application.clearSharedObjects

**Availability**

Flash Communication Server MX.

**Usage**

```
application.clearSharedObjects(soPath);
```

**Parameters**

*soPath* A string that indicates the URI of a shared object.

**Returns**

A Boolean value of `true` if the shared object at the specified path was deleted; otherwise, `false`. If using wildcard characters to delete multiple stream files, the method will return `true` only if all the shared objects matching the wildcard pattern were successfully deleted; otherwise, it will return `false`.

## Description

Method; removes persistent shared objects (FSO files) specified by the *soPath* parameter and clears all properties from active shared objects (both persistent and nonpersistent). You can use the `SharedObject` object to create shared objects in a Flash Communication Server application instance. The Flash Communication Server stores persistent shared objects for each application. You can use `application.clearSharedObjects` to handle one shared object at a time, so you must specify the name of the shared object. You can also use `application.clearStreams` to remove all the recorded streams based on a stream path.

The *soPath* parameter specifies the name of a shared object, which can include a slash (/) as a delimiter between directories in the path. The last element in the path can contain wildcard patterns (for example, a question mark [?]) and an asterisk [\*]) or a shared object name. The `application.clearSharedObjects` method traverses the shared object hierarchy along the specified path and clears all the shared objects. Specifying a slash clears all the shared objects associated with an application instance.

The following are possible values for the *soPath* parameter:

- / clears all local and persistent shared objects associated with the instance.
- /foo/bar clears the shared object /foo/bar; if bar is a directory name, no shared objects are deleted.
- /foo/bar/\* clears all shared objects stored under the instance directory /foo/bar. The bar directory is also deleted if no persistent shared objects are in use within this name space.
- /foo/bar/XX?? clears all shared objects that begin with XX, followed by any two characters. If a directory name matches this specification, all the shared objects within this directory are cleared.

If you call the `clearSharedObjects` method and the specified path matches a shared object that is currently active, all its properties are deleted, and a “clear” event is sent to all subscribers of the shared object. If it is a persistent shared object, the persistent store is also cleared.

## Example

The following example clears all the shared objects for an instance:

```
function onApplicationStop(){
    application.clearSharedObjects("/");
}
```

# Application.clearStreams

## Availability

Flash Communication Server MX.

## Usage

```
application.clearStreams(streamPath);
```

## Parameters

*streamPath* A string that indicates the URI of a stream.

## Returns

A Boolean value of `true` if the stream at the specified path was deleted; otherwise, `false`. If using wildcard characters to clear multiple stream files, the method returns `true` only if all the streams matching the wildcard pattern were successfully deleted; otherwise it returns `false`.

## Description

Method; clears recorded streams (FLV) files and MP3 files associated with the application instance. You can use this method to clear a single stream, all streams associated with the application instance, just those streams in a specific subdirectory of the application instance, or just those streams whose names match a specified wildcard pattern.

The *streamPath* parameter specifies the location and name of a stream, relative to the application's instance directory. You can include a slash (/) as a delimiter between directories in the path. The last element in the path can contain wildcard patterns (for example, a question mark [?] and an asterisk [\*]) or a stream name. The `application.clearStreams` method traverses the stream hierarchy along the specified path and clears all the recorded streams that match the given wildcard pattern. Specifying a slash clears all the streams associated with an application instance.

To clear MP3 files associated with the application instance, precede the path to the stream with `mp3:` (for example, `mp3:/streamPath`). By default, `application.clearStreams` method clears only recorded FLV streams. You can also explicitly clear only FLV streams by placing `flv:` before the stream path (for example, `flv:/streamPath`). See Examples section below for examples of clearing FLV and MP3 files.

The following are examples of some possible values for the *streamPath* parameter:

- `/` or `flv:/` clears all recorded (FLV) streams associated with the instance.
- `/report` clears the stream file `report01.flv` from the application instance directory.
- `/presentations/intro` clears the recorded stream file `intro.flv` from the application instance's `/presentations` subdirectory; if `intro` is a directory name, no streams are deleted.
- `/presentations/*` clears all recorded stream files stored from the application instance's `/presentations` subdirectory. The `/presentation` subdirectory is also deleted if no streams are in use within this name space.
- `mp3:/` clears all MP3 files associated with the instance.
- `mp3:/mozart/requiem` clears the MP3 file named `requiem.mp3` from the application instance's `/mozart` subdirectory.
- `mp3:/mozart/*` clears all MP3 file from the application instance's `/mozart` subdirectory.
- `/presentations/report??` clears all recorded (FLV) streams that begin with `report`, followed by any two characters. If there are directories within the given directory listing, the directories are cleared of any streams that match `report??`.

If an `application.clearStreams` method is invoked on a stream that is currently recording, the recorded file is set to length 0 (cleared), and the internal cached data is also cleared.

**Note:** You can also use the Server Management ActionScript API `removeApp` method to delete all the resources for a single instance.

## Examples

The following example clears all recorded streams:

```
function onApplicationStop(){
    application.clearStreams("/");
}
```

The following example clears all MP3 files from the application instance's `/disco` subdirectory:

```
function onApplicationStop(){
    application.clearStreams("mp3:/disco/*");
}
```

## Application.clients

### Availability

Flash Communication Server MX.

### Usage

`application.clients`

### Description

Property (read-only); an object containing all the Flash clients or other Flash Communication Servers currently connected to the application. The object is a custom object like an array, but with only one property, `length`. Each element in the object is a reference to a `Client` object instance, and you can use the `length` property to determine the number of users connected to the application. You can use the array access operator (`[]`) with the `application.clients` property to access elements in the object.

The object used for the `clients` property is not an array, but it acts the same except for one difference: you can't use the following syntax to iterate through the object:

```
for(var i in application.clients) {
    // insert code here
}
```

Instead, use the following code to loop through each element in a `clients` object:

```
for (var i = 0; i < application.clients.length; i++) {
    // insert code here
}
```

### Example

This example uses a `for` loop to iterate through each member of the `application.clients` array and calls the method `serverUpdate` on each client:

```
for (i = 0; i < application.clients.length; i++){
    application.clients[i].call("serverUpdate");
}
```

## Application.disconnect

### Availability

Flash Communication Server MX.

### Usage

`application.disconnect(clientObj)`

### Parameters

*clientObj* The client to disconnect. The object must be a `Client` object from the `application.clients` array.

### Returns

A Boolean value of `true` if the disconnect was successful; otherwise, `false`.

**Description**

Method; causes the server to terminate a client connection to the application. When this method is called, `NetConnection.onStatus` is invoked on the client side with a status message of `NetConnection.Connection.Closed`. The `application.onDisconnect` method is also invoked.

**Example**

This example calls the `application.disconnect` method to disconnect all users of an application instance:

```
function disconnectAll(){
    for (i=0; i < application.clients.length; i++){
        application.disconnect(application.clients[i]);
    }
}
```

## Application.getStats

**Availability**

Flash Communication Server MX.

**Usage**

`application.getStats()`

**Returns**

An ActionScript object with various properties for each statistic returned.

**Description**

Method; returns statistics for the application instance including the total number of bytes sent and received, the number of RTMP messages sent and received, the number of dropped messages, the number of clients connected to the application instance, and the number of clients who have disconnected from the application instance.

**Example**

The following traces example uses `Application.getStats` to output the application instance's statistics to the Output window:

```
stats = application.getStats();
trace("Total bytes received: " + stats.bytes_in);
trace("Total bytes sent : " + stats.bytes_out);
trace("RTMP messages received : " + stats.msg_in);
trace("RTMP messages sent : " + stats.msg_out);
trace("RTMP messages dropped : " + stats.msg_dropped);
trace("Total clients connected : " + stats.total_connects);
trace("Total clients disconnected : " + stats.total_disconnects);
```

## Application.hostname

**Availability**

- Flash Player 6
- Flash Communication Server MX 1.5

**Usage**

`application.hostname`

**Description**

Property (read-only); contains the hostname of the server for default virtual hosts and the virtual hostname for non-default virtual hosts.

**Example**

The following example traces the name of the host running the current application to the Output window.

```
trace( application.hostname )
```

## Application.name

**Availability**

Flash Communication Server MX.

**Usage**

```
application.name
```

**Description**

Property (read-only); contains the name of the Flash Communication Server application instance.

**Example**

The following example checks the `name` property against a specific string before it executes some code:

```
if (application.name == "videomail/work"){  
    // insert code here  
}
```

## Application.onAppStart

**Availability**

Flash Communication Server MX.

**Usage**

```
application.onAppStart = function (info){  
    // insert code here  
};
```

**Parameters**

None.

**Returns**

Nothing.

**Description**

Event handler; invoked when the server first loads the application instance. You use this handler to initialize an application state. You can use `application.onAppStart` and `application.onAppStop` to initialize and clean up global variables in an application because each of these events is invoked only once during the lifetime of an application instance.

### Example

The following example defines an anonymous function for the `application.onAppStart` event handler that sends a trace message:

```
application.onAppStart = function () {  
    trace ("onAppStart called");  
};
```

## Application.onAppStop

### Availability

Flash Communication Server MX.

### Usage

```
application.onAppStop = function (info){  
    // insert code here  
};
```

### Parameters

*info* An information object that explains why the application stopped running. See Appendix, “Server-Side Information Objects,” on page 77.

### Returns

The value returned by the function you define, if any, or `null`. To refuse to unload the application, return `false`. To unload the application, return `true` or any non-`false` value.

### Description

Event handler; invoked when the application is about to be unloaded by the server. You can define a function that executes when the event handler is invoked. If the function returns `true`, the application unloads. If the function returns `false`, the application doesn't unload. If you don't define a function for this event handler, or if the return value is not a Boolean value, the application is unloaded when the event is invoked.

The Flash Communication Server application passes an information object to the `application.onAppStop` event. You can use server-side ActionScript to look at this information object to decide what to do in the function you define. You could also define the `application.onAppStop` event to notify users before shutdown.

If you use the Administration Console or the Server Management ActionScript API to unload a Flash Communication Server application, `application.onAppStop` is not invoked. Therefore you cannot use the `application.onAppStop` event, for example, to tell users that the application is exiting.

### Example

This example defines a function to perform the shutdown operations on the application. The function is then assigned to the event handler so that it executes when the handler is invoked.

```
function onMyApplicationEnd(info){  
    // Do all the application-specific shutdown logic...  
    // insert code here  
}  
  
application.onAppStop = onMyApplicationEnd;
```

# Application.onConnect

## Availability

Flash Communication Server MX.

## Usage

```
application.onConnect = function (clientObj [, p1, ..., pN]){
    // insert code here to call methods that do authentication
    // returning null puts client in a pending state
    return null;
};
(usage 2)
application.onConnect = function (clientObj [, p1, ..., pN]){
    // insert code here to call methods that do authentication
    // accepts the connection
    application.acceptConnection(clientObj);
};
(usage 3)
application.onConnect = function (clientObj [, p1, ..., pN])
{
    // insert code here to call methods that do authentication
    // accepts the connection by returning true
    return true;
};
```

## Parameters

*clientObj* The client connecting to the application.

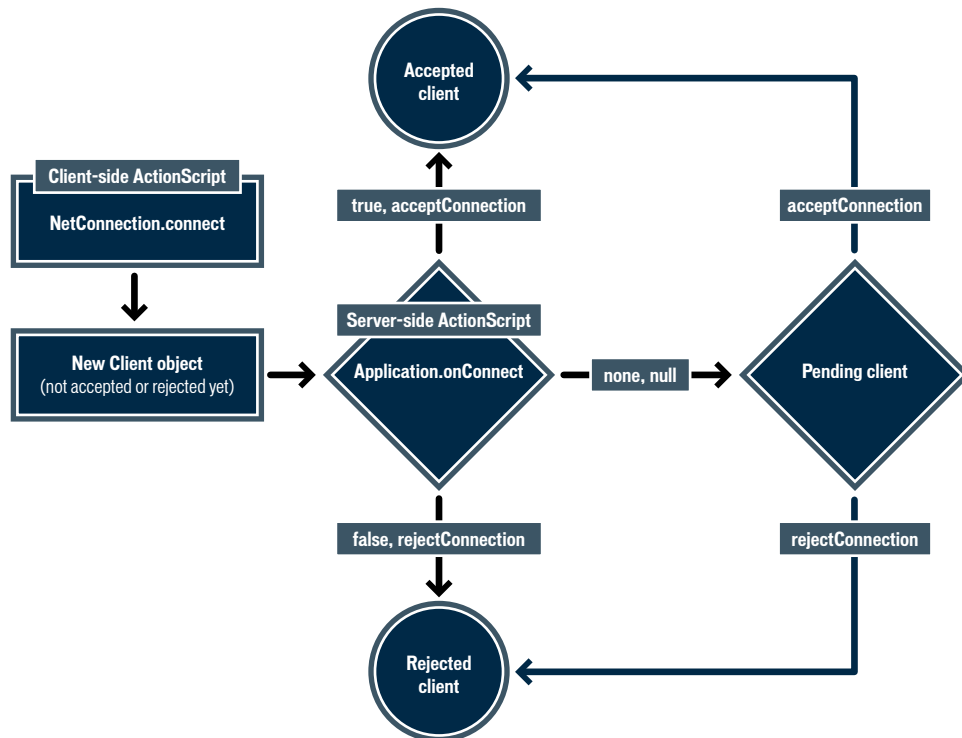
*p1 ..., pN* Optional parameters passed to the `application.onConnect` method. These parameters are passed from the client-side `NetConnection.connect` method when a client connects to the application.

## Returns

The value you provide. If you return a Boolean value of `true`, the server accepts the connection; if the value is `false`, the server rejects the connection. If you return `null` or no return value, the server puts the client in a pending state and the client can't receive or send messages. If the client is put in a pending state, you must call `application.acceptConnection` or `application.rejectConnection` at a later time to accept or reject the connection. For example, you can perform external authentication by making a `NetConnection` call in your `application.onConnect` event handler to an application server, and having the reply handler call `application.acceptConnection` or `application.rejectConnection`, depending on the information received by the reply handler.

You can also call `application.acceptConnection` or `application.rejectConnection` inside the `application.onConnect` event handler. If you do, any value returned by the function is ignored.

**Note:** Returning 1 or 0 is not the same as returning `true` or `false`. The values 1 and 0 are treated the same as any other integers and do not accept or reject a connection.



*How to use `application.onConnect` to accept, reject, or put a client in a pending state*

## Description

Event handler; invoked on the server side when `NetConnection.connect` is called from the client side and a client attempts to connect to an application instance. You can define a function for the `application.onConnect` event handler. If you don't define a function, connections are accepted by default. If the server accepts the new connection, the `application.clients` object is updated.

You can use the `application.onConnect` event in server-side scripts to perform authentication. All the information required for authentication should be sent to the server by the client as parameters (`p1 ...`, `pN`) that you define. In addition to authentication, the script can set the access rights to all server-side objects that this client can modify by setting the `Client.readAccess` and `Client.writeAccess` properties.

If there are several simultaneous connection requests for an application, the server serializes the requests so there is only one `application.onConnect` handler executing at a time. It is a good idea to write code for the `application.onConnect` function that executes quickly to prevent a long connection time for clients.

**Note:** If you are using the Component framework (that is, you are loading the `components.asc` file in your server-side script file) you must use the `Application.onConnectAccept` method to accept client connections. For more information see `Application.onConnectAccept`.

### Example

This example verifies that the user has sent the password “XXXX”. If the password is sent, the user’s access rights are modified and the user can complete the connection. In this case, the user can create or write to streams and shared objects in the user’s own directory and can read or view any shared object or stream in this application instance.

```
// This code should be placed in the global scope

application.onConnect = function (newClient, userName, password){
  // Do all the application-specific connect logic
  if (password == "XXXX"){
    newClient.writeAccess = "/" + userName;
    this.acceptConnection(newClient);
  } else {
    var err = new Object();
    err.message = "Invalid password";
    this.rejectConnection(newClient, err);
  }
};
```

If the password is incorrect, the user is rejected and an information object with a `message` property set to “Invalid password” is returned to the client side. The object is assigned to `infoObject.application`. To access the message property, use the following code on the client side:

```
ClientCom.onStatus = function (info){
  trace(info.application.message);
  // it will print "Invalid password"
  // in the Output window on the client side
};
```

### See also

`Application.acceptConnection`, `Application.onConnectAccept`,  
`Application.onConnectReject`, `Application.rejectConnection`

# Application.onConnectAccept

## Availability

Flash Communication Server MX (with communication components only)

## Usage

```
application.onConnectAccept = function (clientObj [,p1, ..., pN]){  
    //insert code here to indicate result of an accepted connection  
};
```

## Parameters

*clientObj* The client connecting to the application.

*p1...pN* Optional parameters passed to the application.onConnectAccept method. These parameters are passed from the client-side NetConnection.connect method when a client connects to the application.

## Returns

Nothing.

## Description

Event handler; invoked only when the communication components are used (that is, when the components.asc script is loaded into your server-side application script).

application.onConnectAccept is invoked on the server side when NetConnection.connect is called from the client side and a client successfully connects to an application instance.

When you use components, your last line (in order of execution) of the onConnect method should be either application.acceptConnection or application.rejectConnection. Any logic that follows the explicit acceptConnection or rejectConnection statement needs to go into application.onConnectAccept and application.onConnectReject statements, or it will be ignored. This requirement exists only when you use components.

## Example

The first example is the client-side code you would use for an application:

```
nc = new NetConnection();  
  
//try either username  
nc.connect("rtmp:/test","jlopes");  
//nc.connect("rtmp:/test","hacker");  
  
nc.onStatus = function(info) {trace(info.code);}   
  
nc.doSomething = function(){  
    trace("doSomething called!");  
}
```

The second example is server-side code you would include in your application's main.asc file:

```
//When using components, always load components.asc
load( "components.asc" );

application.onConnect = function(client, username){
    trace("onConnect called");
    gFrameworkFC.getClientGlobals(client).username = username;
    if (username == "hacker") {
        application.rejectConnection(client);
    }
    else {
        application.acceptConnection(client);
    }
}

//Code is in onConnectAccept and onConnectReject statements, because
//components are used
application.onConnectAccept = function(client, username){
    trace("Connection accepted for "+username);
    client.call("doSomething",null);
}

application.onConnectReject = function(client, username){
    trace("Connection rejected for "+username);
}
```

#### See also

Application.acceptConnection, Application.onConnect,  
Application.onConnectReject, Application.rejectConnection

## Application.onConnectReject

### Availability

Flash Communication Server MX (with communication components only)

### Usage

```
application.onConnectReject = function (clientObj [,p1, ..., pN]){
    //insert code here to indicate result of a rejected connection
};
```

### Parameters

*clientObj* The client connecting to the application.

*p1...pN* Optional parameters passed to the application.onConnectReject method. These parameters are passed from the client-side NetConnection.connect method when a client connects to the application.

### Returns

Nothing.

### Description

Event handler; invoked only when the communication components are used (that is, only when you include a call to load the components.asc script in your server-side application script). application.onConnectReject is invoked on the server side when NetConnection.connect is called from the client side and a client fails to connect to an application instance.

When you use components, the last line (in order of execution) of the `onConnect` method should be either `application.acceptConnection` or `application.rejectConnection`. Any logic that follows the explicit `acceptConnection` or `rejectConnection` statement needs to go into `application.onConnectAccept` and `application.onConnectReject` statements.

### Example

The first example is the client-side code you would use for an application:

```
nc = new NetConnection();

//try either username
nc.connect("rtmp:/test","jlopes");
//nc.connect("rtmp:/test","hacker");

nc.onStatus = function(info) {trace(info.code);}

nc.doSomething = function(){
    trace("doSomething called!");
}
```

The second example is the server-side code you would include in your `main.asc` file:

```
//When using components, always load components.asc
load( "components.asc" );

application.onConnect = function(client, username){
    trace("onConnect called");
    gFrameworkFC.getClientGlobals(client).username = username;
    if (username == "hacker") {
        application.rejectConnection(client);
    }
    else {
        application.acceptConnection(client);
    }
}

application.onConnectAccept = function(client, username){
    trace("Connection accepted for "+username);
    client.call("doSomething",null);
}

application.onConnectReject = function(client, username){
    trace("Connection rejected for "+username);
}
```

### See also

`Application.acceptConnection`, `Application.onConnect`,  
`Application.onConnectAccept`, `Application.rejectConnection`

## Application.onDisconnect

### Availability

Flash Communication Server MX.

### Usage

```
application.onDisconnect = function (clientObj){
    // insert code here
};
```

**Parameters**

*clientObj* A client disconnecting from the application.

**Returns**

The server ignores any return value.

**Description**

Event handler; invoked when a client disconnects from the application. You can use this event handler to flush any client state information or to notify other users of this event. This event handler is optional.

**Example**

This example uses an anonymous function and assigns it to the `application.onDisconnect` event handler:

```
// This code should be placed in the global scope.
application.onDisconnect = function (client){
    // Do all the client-specific disconnect logic...
    // insert code here
    trace("user disconnected");
};
```

## Application.onStatus

**Availability**

Flash Communication Server MX.

**Usage**

```
application.onStatus = function (infoObject){
    // insert code here
};
```

**Parameters**

*infoObject* An object that contains the error level, code, and sometimes a description. See Appendix, “Server-Side Information Objects,” on page 77.

**Returns**

Any value that the callback function returns.

**Description**

Event handler; invoked when the server encounters an error while processing a message that was targeted at this application instance. The `application.onStatus` event handler is the root for any `Stream.onStatus` or `NetConnection.onStatus` messages that don't find handlers. Also, there are a few status calls that come only to `application.onStatus`. This event handler can be used for debugging messages that generate errors.

**Example**

The following example defines a function that sends a trace statement whenever the `application.onStatus` method is invoked. You can also define a function that gives users specific feedback about the type of error that occurred.

```
appInstance.onStatus = function(infoObject){
    trace("An application error occurred");
};
```

# Application.registerClass

## Availability

Flash Communication Server MX.

## Usage

```
application.registerClass(className, constructor)
```

## Parameters

*className* The name of an ActionScript class.

*constructor* A constructor function used to create an object of a specific class type during object deserialization. The name of the constructor function must be the same as *className*. During object serialization, the name of the constructor function is serialized as the object's type. To unregister the class, pass the value `null` as the *constructor* parameter. Serialization is the process of turning an object into something you can send to another computer over the network.

## Returns

Nothing.

## Description

Method; registers a constructor function that is used when deserializing an object of a certain class type. If the constructor for a class is not registered, you cannot call the deserialized object's methods. This method is also used to unregister the constructor for a class. This is an advanced use of the server and is necessary only when sending ActionScript objects between a client and a server.

The client and the server communicate over a network connection. Therefore, if you use typed objects, each side must have the prototype of the same objects they both use. In other words, both the client-side and server-side ActionScript must define and declare the types of data they share so that there is a clear, reciprocal relationship between an object, method, or property on the client and the corresponding element on the server. You can use `application.registerClass` to register the object's class type on the server side so that you can use the methods defined in the class.

Constructor functions should be used to initialize properties and methods; they should not be used for executing server code. Constructor functions are called automatically when messages are received from the client and need to be "safe" in case they are executed by a malicious client. You shouldn't define procedures that could result in negative situations such as filling up the hard disk or consuming the processor.

The constructor function is called before the object's properties are set. A class can define an `onInitialize` method, which is called after the object has been initialized with all its properties. You can use this method to process data after an object is deserialized.

If you register a class that has its prototype set to another class, you must set the prototype constructor back to the original class after setting the prototype. The second example below illustrates this point.

**Note:** Client-side classes must be defined as `function function_name() {}`, as shown in the following examples. If not defined in the correct way, `application.registerClass` will not properly identify the class when an instance of it is passed from the client to the server, and an error will be returned.

### Example

This example defines a `Color` constructor function with properties and methods. After the application connects, the `registerClass` method is called to register a class for the objects of type `Color`. When a typed object is sent from the client to the server, this class is called to create the server-side object. After the application stops, the `registerClass` method is called again and passes the value `null` to unregister the class.

```
function Color(){
    this.red = 255;
    this.green = 0;
    this.blue = 0;
}
Color.prototype.getRed = function(){
    return this.red;
}
Color.prototype.getGreen = function(){
    return this.green;
}
Color.prototype.getBlue = function(){
    return this.blue;
}
Color.prototype.setRed = function(value){
    this.red = value;
}
Color.prototype.setGreen = function(value){
    this.green = value;
}
Color.prototype.setBlue = function(value){
    this.blue = value;
}
application.onAppStart = function(){
    application.registerClass("Color", Color);
};
application.onAppStop = function(){
    application.registerClass("Color", null);
};
```

The following example shows how to use the `application.registerClass` method with the prototype property:

```
function A(){}
function B(){}

B.prototype = new A();
B.prototype.constructor = B; // set constructor back to that of B
// insert code here
application.registerClass("B", B);
```

## Application.registerProxy

### Availability

Flash Communication Server MX.

### Usage

```
application.registerProxy(methodName, proxyConnection [, proxyMethodName])
```

### Parameters

*methodName* The name of a method. All requests to execute *methodName* for this application instance are forwarded to the *proxyConnection* object.

*proxyConnection* A Client or NetConnection object. All requests to execute the remote method specified by *methodName* are sent to the Client or NetConnection object that is specified in the *proxyConnection* parameter. Any result that is returned is sent back to the originator of the call. To unregister, or remove, the proxy, provide a value of `null` for this parameter.

*proxyMethodName* An optional parameter. The server calls this method on the object specified by the *proxyConnection* parameter if *proxyMethodName* is different from the method specified by the *methodName* parameter.

### Returns

A value that is sent back to the client that made the call.

### Description

Method; maps a method call to another function. You can use this method to communicate between different application instances that can be on the same Flash Communication Server (or different Flash Communication Servers). Clients can execute server-side methods of any application instances to which they are connected. Server-side scripts can use this method to register methods to be proxied to other application instances on the same server or a different server. You can remove or unregister the proxy by calling this method and passing `null` for the *proxyConnection* parameter, which results in the same behavior as never registering the method at all.

### Example

In the following example, the `application.registerProxy` method is called in a function in the `application.onAppStart` event handler and executes when the application starts. Inside the function block, a new `NetConnection` object called `myProxy` is created and connected. The `application.registerProxy` method is then called to assign the method `getXYZ` to the `myProxy` object.

```
application.onAppStart = function(){
    var myProxy = new NetConnection();
    myProxy.connect("rtmp://xyz.com/myApp");
    application.registerProxy("getXYZ", myProxy);
};
```

## Application.rejectConnection

### Availability

Flash Communication Server MX.

### Usage

```
application.rejectConnection(clientObj, errObj)
```

### Parameters

*clientObj* A client to reject.

*errObj* An object of any type that is sent to the client, explaining the reason for rejection. The *errObj* object is available in client-side scripts as the `application` property of the information object that is passed to the `application.onStatus` call when the connection is rejected. For more information, see the Appendix, “Client-Side Information Objects,” in the *Client-Side Communication ActionScript Dictionary*.

### Returns

Nothing.

## Description

Method; rejects the connection call from a client to the server. The `application.onConnect` event handler notifies a script when a new client is connecting. In the function assigned to `application.onConnect`, you can either accept or reject the connection. You can also define a function for `application.onConnect` that calls an application server for authentication. In that case, an application could call `application.rejectConnection` from the application server's response callback to disconnect the client from the server.

When you use components and your code includes an explicit call to `application.acceptConnection` or `application.rejectConnection`, the last line (in order of execution) of the `onConnect` method should be either `application.acceptConnection` or `application.rejectConnection`. Also, any logic that follows the explicit `acceptConnection` or `rejectConnection` statement must be placed in `application.onConnectAccept` and `application.onConnectReject` statements, or it will be ignored. This requirement exists only when you use components.

## Example

In the following example, the client specified by `client1` is rejected and provided with the error message contained in `err.message`. The message "Too many connections" appears on the server side.

```
function onConnect(client1){
    // insert code here
    var err = new Object();
    err.message = "Too many connections";
    application.rejectConnection(client1, err);
}
```

The following code should appear on the client side:

```
clientConn.onStatus = function (info){
    if (info.code == "NetConnection.Connect.Rejected"){
        trace(info.application.message);
        // this sends the message
        // "Too many connections" to the Output window
        // on the client side
    }
};
```

## See also

`Application.onConnect`, `Application.acceptConnection`

# Application.server

## Availability

Flash Communication Server MX.

## Usage

`application.server`

## Description

Property (read-only); contains the platform and the server-version information.

### Example

The following example checks the `server` property against a string before executing the code inside the `if` statement:

```
if (application.server == "Flash Communication Server-Windows/1.0"){  
    // insert code here  
}
```

## clearInterval

### Availability

Flash Communication Server MX.

### Usage

```
clearInterval(intervalID)
```

### Parameters

*intervalID* A unique ID returned by a previous call to the `setInterval` method.

### Returns

Nothing.

### Description

Method (global); cancels a time-out that was set with a call to the `setInterval` method.

### Example

The following example creates a function named `callback` and passes it to the `setInterval` method, which is called every 1000 milliseconds and sends the message “interval called” to the Output window. The `setInterval` method returns a unique identifier that is assigned to the variable `intervalID`. The identifier allows you to cancel a specific `setInterval` call. In the last line of code, the `intervalID` variable is passed to the `clearInterval` method to cancel the `setInterval` call:

```
function callback(){  
    trace("interval called");  
}  
var intervalID;  
intervalID = setInterval(callback, 1000);  
// sometime later  
clearInterval(intervalID);
```

## Client (object)

The Client object lets you handle each user, or *client*, connection to a Flash Communication Server application instance. The server automatically creates a Client object when a user connects to an application; the object is destroyed when the user disconnects from the application. Users have unique Client objects for each application to which they are connected. Thousands of Client objects can be active at the same time.

You can use the properties of the Client object to determine the version, platform, and IP address of each client. You can also set individual read and write permissions to various application resources such as Stream objects and shared objects. Use the methods of the Client object to set bandwidth limits and call methods in client-side scripts.

When you call `NetConnection.call` from a client-side ActionScript script, the method that executes in the server-side script must be a method of the Client object. In your server-side script, you must define any method that you want to call from the client-side script. You can also call any methods you define in the server-side script directly from the Client object instance in the server-side script.

If all instances of the Client object (each client in an application) require the same methods or properties, you can add those methods and properties to the class itself instead of adding them to each instance of a class. This process is called *extending* a class. You can extend any server-side or client-side ActionScript class. To extend a class, instead of defining methods inside the constructor function of the class or assigning them to individual instances of the class, you assign methods to the `prototype` property of the constructor function of the class. When you assign methods and properties to the `prototype` property, the methods are automatically available to all instances of the class.

Extending a class lets you define the methods of a class separately, which makes them easier to read in a script. And more importantly, it is more efficient to add methods to `prototype`, otherwise the methods are reinterpreted each time an instance is created.

The following code shows how to assign methods and properties to an instance of a class. During `application.onConnect`, a client instance `clientObj` is passed to the server-side script as a parameter. You can then assign a property and method to the client instance:

```
application.onConnect = function(clientObj){
    clientObj.birthday = myBDay;
    clientObj.calculateDaysUntilBirthday = function(){
        // insert code here
    }
};
```

The above example works, but must be executed every time a client connects. If you want the same methods and properties to be available to all clients in the `application.clients` array without defining them every time, you must assign them to the `prototype` property of the `Client` object. There are two steps to extending a built-in class using the `prototype` method. You can write the steps in any order in your script. The following example extends the built-in `Client` object, so the first step is to write the function that you will assign to the `prototype` property:

```
// first step: write the functions

function Client_getWritePermission(){
// "writeAccess" property is already built-in to the client class
return this.writeAccess;
}

function Client_createUniqueID(){
    var ipStr = this.ip;
// "ip" property is already built-in to the client class
    var uniqueID = "rel23mn"
// you would need to write code in the above line
// that creates a unique ID for each client instance
    return uniqueID;
}

// second step: assign prototype methods to the functions

Client.prototype.getWritePermission = Client_getWritePermission;
Client.prototype.createUniqueID = Client_createUniqueID;

// a good naming convention is to start all class method
// names with the name of the class, followed by an underscore
```

You can also add properties to prototype, as shown in the following example:

```
Client.prototype.company = "Macromedia";
```

The methods are available to any instance, so within `application.onConnect`, which is passed a `clientObj` argument, you can write the following code:

```
application.onConnect = function(clientObj){
    var clientID = clientObj.createUniqueID();
    var clientWritePerm = clientObj.getWritePermission();
};
```

## Method summary for the Client object

Method	Description
<code>Client.call</code>	Executes a method on the Flash client asynchronously and returns the value from the Flash client to the server.
<code>Client.getBandwidthLimit</code>	Returns the maximum bandwidth the client or the server can attempt to use for this connection.
<code>Client.getStats</code>	Returns statistics for the client.
<code>Client.ping</code>	Sends a "ping" message to the client. If the client responds, the method returns true; otherwise it returns false.
<code>Client.__resolve</code>	Provides values for undefined properties.
<code>Client.setBandwidthLimit</code>	Sets the maximum bandwidth for the connection.

## Property summary for the Client object

Property	Description
<code>Client.agent</code>	The version and platform of the Flash client.
<code>Client.ip</code>	The IP address of the Flash client.
<code>Client.protocol</code>	The protocol used by the client to connect to the server.
<code>Client.readAccess</code>	A list of access levels to which the client has read access.
<code>Client.referrer</code>	The URL of the SWF file or server where this connection originated.
<code>Client.writeAccess</code>	A list of access levels to which the client has write access.

## Event handler summary for the Client object

Event handler	Description
<code>Client."commandName"</code>	Invoked when <code>NetConnection.call(commandName)</code> is called in a client-side script.

## Client.agent

### Availability

Flash Communication Server MX.

### Usage

`Client.agent`

### Description

Property (read-only); contains the version and platform information of the Flash client.

### Example

The following example checks the `agent` property against the string "WIN" and executes different code depending on whether they match. This code is written inside an `onConnect` function.

```
function onConnect(newClient, name){
    if (newClient.agent.indexOf("WIN") > -1){
        trace ("Window user");
    } else {
        trace ("non Window user.agent is" + newClient.agent);
    }
}
```

## Client.call

### Availability

Flash Communication Server MX.

### Usage

```
Client.call(methodName, [resultObj, [p1, ..., pN]])
```

### Parameters

*methodName* A method specified in the form `[objectPath/]method`. For example, the command `someObj.doSomething` tells the client to invoke the `NetConnection.someObj.doSomething` method on the client.

*resultObj* An optional parameter that is required when the sender expects a return value from the client. If parameters are passed but no return value is desired, pass the value `null`. The result object can be any object you define and, in order to be useful, should have two methods that are invoked when the result arrives: `onResult` and `onStatus`. The `resultObj.onResult` event is triggered if the invocation of the remote method is successful; otherwise, the `resultObj.onStatus` event is triggered.

*p1*, ..., *pN* Optional parameters that can be of any `ActionScript` type, including a reference to another `ActionScript` object. These parameters are passed to the *methodName* parameter when the method executes on the Flash client. If you use these optional parameters, you must pass in some value for *resultObj*; if you do not want a return value, pass `null`.

### Returns

A Boolean value of `true` if a call to *methodName* was successful on the client; otherwise, `false`.

### Description

Method; executes a method on the originating Flash client or on another server. The remote method may optionally return data, which is returned as a result to the *resultObj* parameter, if it is provided. The remote object is typically a Flash client connected to the server, but it can also be another server. Whether the remote agent is a Flash client or another server, the method is called on the remote agent's `NetConnection` object.

### Example

The following example shows a client-side script that defines a function called `random`, which generates a random number:

```
nc = new NetConnection();
nc.connect ("rtmp://someserver/someApp/someInst");
nc.random = function(){
    return (Math.random());
};
```

The following server-side script uses the `Client.call` method inside the `application.onConnect` handler to call the `random` method that was defined on the client side. The server-side script also defines a function called `randHandler`, which is used in the `Client.call` method as the *resultObj* parameter.

```
application.onConnect = function(clientObj){
    trace("we are connected");
    application.acceptConnection(clientObj);
    clientObj.call("random", new randHandler());
};
randHandler = function(){
    this.onResult = function(res){
        trace("random num: " + res);
    }
    this.onStatus = function(info){
        trace("failed and got:" + info.code);
    }
};
```

## Client."commandName"

### Availability

Flash Communication Server MX.

### Usage

```
function onMyClientCommand([p1, ..., pN])
{
    // insert code here
    return val;
}
```

### Parameters

*p1, ..., pN* Optional parameters passed to the command message handler if the message contains parameters you defined. These parameters are the `ActionScript` objects passed to the `NetConnection.call` method.

### Returns

Any `ActionScript` object you define. This object is serialized and sent to the client as a response to the command only if the command message supplied a response handler.

### Description

Event handler; invoked when a Flash client or another server calls the `NetConnection.call` method. A command name parameter is passed to `NetConnection.call` on the client side, which causes Flash Communication Server to search the `Client` object instance on the server side for a function that matches the command name parameter. If the parameter is found, the method is invoked and the return value is sent back to the client.

### Example

This example creates a method called `sum` as a property of the `Client` object `newClient` on the server side:

```
newClient.sum = function sum(op1, op2){
    return op1 + op2;
};
```

The `sum` method can then be called from `NetConnection.call` on the Flash client side, as shown in the following example:

```
nc = new NetConnection();
nc.connect("rtmp://myServer/myApp");
nc.call("sum", new result(), 20, 50);
function result(){
    this.onResult = function (retVal){
        output += "sum is " + retVal;
    };
    this.onStatus = function(errorVal){
        output += errorVal.code + " error occurred";
    };
}
```

The `sum` method can also be called on the server side, as shown here:

```
newClient.sum();
```

The following example creates two functions that you can call from either a client-side or server-side script:

```
application.onConnect = function(clientObj) {
    // Add a callable function called "foo"; foo returns the number 8
    clientObj.foo = function() {return 8;};
    // Add a remote function that is not defined in the onConnect call
    clientObj.bar = application.barFunction;
};
// The bar function adds the two values it is given
application.barFunction = function(v1,v2) {
    return (v1 + v2);
};
```

You can call either of the two functions that were defined in the previous examples (`foo` and `bar`) by using the following code in a client-side script:

```
c = new NetConnection();
c.call("foo");
c.call("bar", null, 1, 1);
```

You can call either of the two functions that were defined in the previous examples (`foo` and `bar`) by using the following code in a server-side script:

```
c = new NetConnection();
c.onStatus = function(info) {
    if(info.code == "NetConnection.Connect.Success") {
        c.call("foo");
        c.call("bar", null, 2, 2);
    }
};
```

## Client.getBandwidthLimit

### Availability

Flash Communication Server MX.

### Usage

```
Client.getBandwidthLimit(iDirection)
```

### Parameters

*iDirection* An integer specifying the connection direction: 0 indicates a client-to-server direction, 1 indicates a server-to-client direction.

**Returns**

An integer indicating bytes per second.

**Description**

Method; returns the maximum bandwidth that the client or the server can use for this connection. Use the *iDirection* parameter to get the value for each direction of the connection. The value returned indicates bytes per second and can be changed with `Client.setBandwidthLimit`. The default value for a connection is set for each application in the `Application.xml` file.

**Example**

The following example uses `Client.getBandwidthLimit` with the *iDirection* parameter to set two variables, `clientToServer` and `serverToClient`:

```
application.onConnect = function(newClient){
    var clientToServer= newClient.getBandwidthLimit(0);
    var serverToClient= newClient.getBandwidthLimit(1);
};
```

## Client.getStats

**Availability**

Flash Communication Server MX.

**Usage**

```
Client.getStats()
```

**Returns**

An object with various properties for each statistic returned.

**Description**

Method; returns statistics for the client including the total number of bytes sent and received, the number of RTMP messages sent and received, the number of dropped RTMP messages, and how long it takes the client takes to respond to a ping message.

**Example**

The following example uses `Client.getStats` to output a new client's statistics to the Output window:

```
application.onConnect( newClient ){
    stats = newClient.getStats();
    trace("Total bytes received : " + stats.bytes_in);
    trace("Total bytes sent : " + stats.bytes_out);
    trace("RTMP messages received : " + stats.msg_in);
    trace("RTMP messages sent: " + stats.msg_out);
    trace("RTMP messages dropped : " + stats.msg_dropped);
    trace("Ping roundtrip time: " + stats.ping_rtt);
}
```

**See also**

```
Client.ping
```

## Client.ip

### Availability

Flash Communication Server MX.

### Usage

`Client.ip`

### Description

Property (read-only); contains the IP address of the Flash client.

### Example

The following example uses the `Client.ip` property to verify whether a new client has a specific IP address. The result determines which block of code runs.

```
application.onConnect = function(newClient, name){
    if (newClient.ip == "127.0.0.1"){
        // insert code here
    } else {
        // insert code here
    }
};
```

## Client.protocol

### Availability

Flash Communication Server MX

### Usage

`Client.protocol`

### Description

Property (read-only); contains a string that indicates the protocol used by the client to connect to the server. This string can have one of the following values:

- `rtmp` (Real Time Message Protocol over persistent socket connection)
- `rtmpt` (RTMP tunneled through HTTP protocol)

For more information about the HTTP tunneling feature in Flash Communication Server MX see the `NetConnection.connect` entry in the *Client-Side Communication ActionScript Dictionary*.

### Example

The following checks the connection protocol used by a client upon connection to the application.

```
application.onConnect(clientObj) {
    if(clientObj.protocol == "rtmp") {
        trace("Client connected over a persistent connection");
    } else if(clientObj.protocol == "rtmpt") {
        trace("Client connected over an HTTP tunneling connection");
    }
}
```

## Client.ping

### Availability

Flash Communication Server MX.

### Usage

`Client.ping`

### Description

Method; sends a "ping" message to the client and waits for a response. If the client responds, the method returns `true`, otherwise returns `false`. Use this method to determine whether or not the client connection is still alive and active.

### Example

The following `onConnect` function pings the connecting client and traces the results of the method to the Output window:

```
application.onConnect( newClient ) {  
    if ( newClient.ping() ){  
        trace("ping successful");  
    }  
    else {  
        trace("ping failed");  
    }  
}
```

### See also

`Client.getStats`

## Client.readAccess

### Availability

Flash Communication Server MX.

### Usage

`Client.readAccess`

### Description

Property; provides read-access rights to directories that contain application resources (shared objects and streams) for this client. To give a client read access to directories containing application resources, list directories in a string delimited by semicolons.

By default, all clients have full read access, and the `readAccess` property is set to slash (/). To give a client read access, specify a list of access levels (in URI format), delimited by semicolons. Any files or directories within a specified URI are also considered accessible. For example, if `myMedia` is specified as an access level, then any files or directories in the `myMedia` directory are also accessible (for example, `myMedia/mp3s`). Similarly, any files or directories in the `myMedia/mp3s` directory are also accessible, and so on.

Clients with read access to a directory that contains streams can play streams in the specified access levels. Clients with read access to a directory that contains shared objects can subscribe to shared objects in the specified access levels and receive notification of changes in the shared objects.

- For streams, `readAccess` controls which streams can be played by the connection.
- For shared objects, `readAccess` controls if the connection can listen to shared object changes.

**Tip:** Although you cannot use this property to control access for a particular file, you can create a separate directory for a file if you want to control access to it.

### Example

The following `onConnect` function gives a client read access to `myMedia/mp3s`, `myData/notes`, and any files or directories within them:

```
application.onConnect = function(newClient, name){
    newClient.readAccess = "myMedia/mp3s;myData/notes";
};
```

### See also

`Client.writeAccess`

## Client.referrer

### Availability

Flash Communication Server MX.

### Usage

`Client.referrer`

### Description

Property (read-only); a string object whose value is set to the URL of the SWF file or the server in which this connection originated.

### Example

The following example defines an `onConnect` callback function that sends a trace to the Output window indicating the origin of the new client when that client connects to the application:

```
application.onConnect = function(newClient, name){
    trace("New user connected to server from" + newClient.referrer);
};
```

## Client.\_\_resolve

### Availability

Flash Communication Server MX.

### Usage

```
Client.__resolve = function(propName){
    // insert code here
};
```

### Parameters

*propName* The name of an undefined property.

### Returns

The value of the undefined property, which is specified by the *propName* parameter.

### Description

Method; provides values for undefined properties. When an undefined property of a Client object is referenced by server-side ActionScript code, that object is checked for a `__resolve` method. If the object has a `__resolve` method, the `__resolve` method is invoked and passed the name of the undefined property. The return value of the `__resolve` method is the value of the undefined property. In this way, `__resolve` can supply the values for undefined properties and make it appear as if they are defined.

### Example

The following example defines a function that is called whenever an undefined property is referenced:

```
Client.prototype.__resolve = function (name) {  
    return "Hello, world!";  
};  
function onConnect(newClient){  
    trace (newClient.property1); // this will print "Hello World"  
}
```

## Client.setBandwidthLimit

### Availability

Flash Communication Server MX.

### Usage

```
Client.setBandwidthLimit(iServerToClient, iClientToServer)
```

### Parameters

*iServerToClient* The bandwidth from server to client, in bytes per second. Use 0 if you don't want to change the current setting.

*iClientToServer* The bandwidth from client to server, in bytes per second. Use 0 if you don't want to change the current setting.

### Returns

Nothing.

### Description

Method; sets the maximum bandwidth for this client from client to server, server to client, or both. The default value for a connection is set for each application in the Application.xml file. The value specified cannot exceed the bandwidth cap value specified in the Application.xml file.

### Example

The following example sets the bandwidth limits for each direction, based on values passed to the `onConnect` function:

```
application.onConnect = function(newClient, serverToClient, clientToServer){  
    newClient.setBandwidthLimit(serverToClient, clientToServer);  
    application.acceptConnection(newClient);  
}
```

## Client.writeAccess

### Availability

Flash Communication Server MX.

### Usage

`Client.writeAccess`

### Description

Property; provides write-access rights to directories containing application resources (such as shared objects and streams) for this client. To give a client write access to directories containing application resources, list directories in a string delimited by semicolons. By default, all clients have full write access, and the `writeAccess` property is set to slash (/). For example, if `myMedia` is specified as an access level, then any files or directories in the `myMedia` directory are also accessible (for example, `myMedia/myStreams`). Similarly, any files or subdirectories in the `myMedia/myStreams` directory are also accessible, and so on.

- For shared object, `writeAccess` provides control over who can create and update the shared objects.
- For streams, `writeAccess` provides control over who can publish and record a stream.

**Tip:** Although you cannot use this property to control access for a particular file, you can create a separate directory for a file if you want to control access to it.

### Example

The following example provides write access to the `/myMedia/myStreams` and `myData/notes` directories.

```
application.onConnect = function(newClient, name){
    newClient.writeAccess = "/myMedia/myStreams;myData/notes";
    application.acceptConnection();
};
```

### See also

`Client.readAccess`

## load

### Availability

Flash Communication Server MX.

### Usage

`load(filename);`

### Parameters

*filename* The relative path to an ActionScript file in relation to the main.asc file.

### Returns

Nothing.

### Description

Method (global); loads an ActionScript file inside the main.asc file. This method executes only when the ActionScript file is first loaded. The loaded file is compiled and executed after the main.asc file is successfully loaded, compiled, and executed, and before `application.onAppStart` is executed. The path of the specified file is resolved relative to main.asc. This method is useful for loading ActionScript libraries.

**Note:** For security reasons, your server-side applications directory, which contains ASC files, audio/video FLV files, and ActionScript FLA source files, should not be accessible to a user browsing your website.

### Example

The following example loads the myLoadedFile.as file:

```
load("myLoadedFile.as");
```

## NetConnection (object)

The server-side NetConnection object lets you create a two-way connection between a Flash Communication Server application instance and an application server, another Flash Communication Server, or another Flash Communication Server application instance on the same server. You can use NetConnection objects to create powerful applications; for example, you can get weather information from an application server or share an application load with other Flash Communication Servers or application instances.

You can use Macromedia Flash Remoting with the Flash Communication Server to communicate with application servers such as Macromedia ColdFusion MX, Macromedia JRun, Microsoft .NET, and J2EE servers using Action Message Format (AMF) over HTTP. AMF lets you transfer light-weight binary data between a Flash Communication Server or a Flash client and a web application server. For more information, see [www.macromedia.com/go/flashremoting](http://www.macromedia.com/go/flashremoting).

You can also connect to an application server for server-to-server interactions using standard protocols (such as HTTP) or connect to another Flash Communication Server for sharing audio, video, and data using the Macromedia Real-Time Messaging Protocol (RTMP).

To create a NetConnection object, use the constructor function described below.

## Method summary for the NetConnection object

Method	Description
<code>NetConnection.addHeader</code>	Adds a context header.
<code>NetConnection.call</code>	Invokes a method or operation on a remote server.
<code>NetConnection.close</code>	Closes a server connection.
<code>NetConnection.connect</code>	Establishes connection to a server.

## Property summary for the NetConnection object

Property	Description
<code>NetConnection.isConnected</code>	A Boolean value indicating whether a connection has been made.
<code>NetConnection.uri</code>	The URI that was passed by the <code>NetConnection.connect</code> method.

## Event handler summary for the NetConnection object

Event handler	Description
<code>NetConnection.onStatus</code>	Called when there is a change in connection status.

## Constructor for the NetConnection object

### Availability

Flash Communication Server MX.

### Usage

```
new NetConnection()
```

### Parameters

None.

### Returns

A `NetConnection` object.

### Description

Constructor; creates a new instance of the `NetConnection` object.

### Example

The following example creates a new instance of the `NetConnection` object:

```
newNC = new NetConnection();
```

## NetConnection.addHeader

### Availability

Flash Communication Server MX.

### Usage

```
myNetConn.addHeader(name, mustUnderstand, object)
```

### Parameters

*name* A string that identifies the header and the ActionScript object data associated with it.

*mustUnderstand* A Boolean value; *true* indicates that the server must understand and process this header before it handles any of the following headers or messages.

*object* Any ActionScript object.

### Returns

Nothing.

### Description

Method; adds a context header to the AMF packet structure. This header is sent with every future AMF packet. If you call `NetConnection.addHeader` using the same name, the new header replaces the existing header, and the new header persists for the duration of the `NetConnection` object. You can remove a header by calling `NetConnection.addHeader` with the name of the header to remove and an undefined object.

### Example

The following example creates a new `NetConnection` instance, *nc*, and connects to an application at web server `www.foo.com` that is listening at port 1929. This application dispatches the service `/blag/SomeCoolService`. The last line of code adds a header called `foo`:

```
nc=new NetConnection();
nc.connect("http://www.foo.com:1929/blag/SomeCoolService");
nc.addHeader("foo", true, new Foo());
```

## NetConnection.call

### Availability

Flash Communication Server MX.

### Usage

```
myNetConnection.call(methodName, [resultObj, p1, ..., pN])
```

### Parameters

*methodName* A method specified in the form `[objectPath/]method`. For example, the command `someObj.doSomething` tells the remote server to invoke the `clientObj.someObj.doSomething` method, with all the *p1, ..., pN* parameters. If the object path is missing, `clientObj.doSomething()` is invoked on the remote server.

*resultObj* An optional parameter that is used to handle return values from the server. The result object can be any object you defined and can have two defined methods to handle the returned result: `onResult` and `onStatus`. If an error is returned as the result, `onStatus` is invoked; otherwise, `onResult` is invoked.

*p1, ..., pN* Optional parameters that can be of any ActionScript type, including a reference to another ActionScript object. These parameters are passed to the *methodName* specified above when the method is executed on the remote application server.

### Returns

For RTMP connections, returns a Boolean value of *true* if a call to *methodName* is sent to the client; otherwise, *false*. For application server connections, it always returns *true*.

### Description

Method; invokes a command or method on a Flash Communication Server or an application server to which the application instance is connected. The `NetConnection.call` method on the server works the same way as the `NetConnection.call` method on the client: it invokes a command on a remote server.

**Note:** If you want to call a method on a client from a server, use the `Client.call` method.

### Example

This example uses RTMP to execute a call from one Flash Communication Server to another Flash Communication Server. The code makes a connection to the `App1` application on server 2 and then invokes the method `Sum` on server 2:

```
nc1.connect("rtmp://server2.mydomain.com/App1", "svr2",);  
nc1.call("Sum", new Result(), 3, 6);
```

The following server-side ActionScript code is on server 2. When the client is connecting, this code checks to see whether it has an argument that is equal to `svr1`. If the client has that argument, the `Sum` method is defined so that when the method is called from `svr1`, `svr2` can respond with the appropriate method:

```
application.onConnect = function(clientObj){  
    if(arg1 == "svr1"){  
        clientObj.Sum = function(p1, p2){  
            return p1 + p2;  
        }  
    }  
    return true;  
};
```

The following example uses an AMF request to make a call to an application server. This allows Flash Communication Server to connect to an application server and then invoke the `quote` method. The Java adaptor dispatches the call by using the identifier to the left of the dot as the class name and the identifier to the right of the dot as a method of the class.

```
nc = new NetConnection;  
nc.connect("http://www.xyz.com/java");  
nc.call("myPackage.quote", new Result());
```

## NetConnection.close

### Availability

Flash Communication Server MX.

### Usage

```
myNetConnection.close()
```

### Parameters

None.

### Returns

Nothing.

### Description

Method; closes the connection with the server. After you close the connection, you can reuse the `NetConnection` instance and reconnect to an old application or connect to a new one.

**Note:** The `NetConnection.close` method has no effect on HTTP connections.

### Example

The following code closes the `NetConnection` instance `myNetConn`:

```
myNetConn.close();
```

## NetConnection.connect

### Availability

Flash Communication Server MX.

### Usage

```
myNetConnection.connect(URI, [p1, ..., pN])
```

### Parameters

*URI*    A URI to connect to.

*p1*, ..., *pN*    Optional parameters that can be of any `ActionScript` type, including references to other `ActionScript` objects. These parameters are sent as connection parameters to the `application.onConnect` event handler for RTMP connections. For AMF connections to application servers, any RTMP parameters are ignored.

### Returns

For RTMP connections, a Boolean value of `true` for success; `false` otherwise. For AMF connections to application servers, `true` is always returned.

### Description

Method; connects to the host. The host URI has the following format:

*[protocol://]host[:port]/appName[/instanceName]*

For example, the following are legal URIs:

```
http://appServer.mydomain.com/webApp  
rtmp://rtserver.mydomain.com/realtimeApp
```

It is good practice to write an `application.onStatus` callback function and check the `NetConnection.isConnected` property for RTMP connections to see whether a successful connection was made. For Action Message Format connections, check `NetConnection.onStatus`.

### Example

This example creates an RTMP connection to another Flash Communication Server for the `myConn` instance of `NetConnection`:

```
myConn = new NetConnection();  
myConn.connect("rtmp://tc.foo.com/myApp/myConn");
```

The following example creates an AMF connection to an application server for the `myConn` instance of `NetConnection`:

```
myConn = new NetConnection();  
myConn.connect("http://www.xyz.com/myApp/");
```

# NetConnection.isConnected

## Availability

Flash Communication Server MX.

## Usage

```
myNetConnection.isConnected
```

## Description

Property (read-only); a Boolean value that indicates whether a connection has been made. It is set to true if there is a connection to the server. It's a good idea to check this property value in the `onStatus` callback function. This property is always true for AMF connections to application servers.

## Example

This example uses `NetConnection.isConnected` inside an `onStatus` definition to check if a connection has been made:

```
nc = new NetConnection();
nc.connect("rtmp://tc.foo.com/myApp");
nc.onStatus = function(infoObj){
    if (info.code == "NetConnection.Connect.Success" && nc.isConnected){
        trace("We are connected");
    }
};
```

# NetConnection.onStatus

## Availability

Flash Communication Server MX.

## Usage

```
myNetConnection.onStatus = function(infoObject) {
    // Your code here
};
```

## Parameters

*infoObject* An information object. For details about this parameter, see Appendix, “Server-Side Information Objects,” on page 77.

## Returns

Nothing.

## Description

Event handler; invoked every time the status of the `NetConnection` object changes. For example, if the connection with the server is lost in an RTMP connection, the

`NetConnection.isConnected` property is set to false, and `NetConnection.onStatus` is invoked with a status message of `NetConnection.Connect.closed`. For AMF connections, `NetConnection.onStatus` is used only to indicate a failed connection. Use this event handler to check for connectivity.

### Example

This example defines a function for the `onStatus` handler that outputs messages to indicate whether the `NetConnection` was successful:

```
nc = new NetConnection();
nc.onStatus = function(info){
    if (info.code == "NetConnection.Connect.Success") {
        _root.gotoAndStop(2);
    } else {
        if (! nc.isConnected){
            _root.gotoAndStop(1);
        }
    }
};
```

## NetConnection.uri

### Availability

Flash Communication Server MX.

### Usage

*myNetConnection.uri*

### Description

Property (read-only); A string indicating the URI that was passed by the `NetConnection.connect` method. This property is set to `null` before a call to `NetConnection.connect` or after `NetConnection.close` is called.

## setInterval

### Availability

Flash Communication Server MX.

### Usage

```
setInterval(function, interval[, p1, ..., pN]);
setInterval(object, methodName, interval[, p1, ..., pN]);
```

### Parameters

*function* The name of a defined ActionScript function or a reference to an anonymous function.

*object* An object derived from the ActionScript Object object.

*methodName* The name of the method to call on *object*.

*interval* The time (interval) between calls to *function*, in milliseconds.

*p1*, ..., *pN* Optional parameters passed to *function*.

### Returns

A unique ID for this call. If the interval is not set, the method returns -1.

## Description

Method (global); continually calls a function or method at a specified time interval until the `clearInterval` method is called. This method allows a server-side script to run a generic routine. The `setInterval` method returns a unique ID that you can pass to the `clearInterval` method to stop the routine.

**Note:** Standard JavaScript supports an additional usage for the `setInterval` method, `setInterval(stringToEvaluate, timeInterval)`, which is not supported by Server-Side Communication ActionScript.

## Example

The following example uses an anonymous function to send the message “interval called” to the server log every second:

```
setInterval(function(){trace("interval called");}, 1000);
```

The following example also uses an anonymous function to send the message “interval called” to the server log every second, but it passes the message to the function as a parameter:

```
setInterval(function(s){trace(s);}, 1000, "interval called");
```

The following example uses a named function, `callback1`, to send the message “interval called” to the server log:

```
function callback1(){
    trace("interval called");
}
setInterval(callback1, 1000);
```

The following example also uses a named function, `callback2`, to send the message “interval called” to the server log, but it passes the message to the function as a parameter:

```
function callback2(s){
    trace(s);
}
setInterval(callback2, 1000, "interval called");
```

## SharedObject (object)

Shared objects let you share data between multiple client movies in real time. Shared objects can be persistent on the server and you can think of these objects as real-time data transfer devices. You can also use the client-side ActionScript `SharedObject` object to create shared objects on the client. For more information, see the `SharedObject` entry in the *Client-Side Communication ActionScript Dictionary*. The following list describes common ways to use remote shared objects:

- Storing and sharing data on a server

A shared object can store data on the server for other clients to retrieve. For example, you can open a remote shared object, such as a phone list, that is persistent on the server. Whenever a client makes a change to the shared object, the revised data is available to all clients that are currently connected to the object or that connect to it later. If the object is also persistent locally and a client changes the data while not connected to the server, the changes are copied to the remote shared object the next time the client connects to the object.

- Sharing data in real time

A shared object can share data among multiple clients in real time. For example, you can open a remote shared object that stores real-time data (such as a list of users connected to a chat room) that is visible to all clients connected to the object. When a user enters or leaves the chat room, the object is updated and all clients that are connected to the object see the revised list of chat room users.

Every shared object is identified by a unique name and contains a list of name-value pairs, called *properties*, just like any other ActionScript object. A name must be a unique string and a value can be any ActionScript data type. (For more information about data types, see *Using Flash MX*.) All shared objects have a data property. Any property of the data property can be shared and is called a *slot*.

A shared object can be owned by the current (local) application instance or by a different (remote) application instance. The remote application instance can be on the same server or on a different server. References to shared objects that are owned by a remote application instance are called *proxied shared objects*.

The slot of a shared object owned by the local instance can be modified by multiple clients or by the server simultaneously; there is no conflict on the server side when a shared object is modified. For example, a call to `SharedObject.getProperty` returns the latest value and setting a new value assigns a new value for the named slot and updates the object version. If you write a server-side script that modifies multiple properties, you can prevent other clients from modifying the object during the update, by calling the `SharedObject.lock` method before updating the object. Then you can call `SharedObject.unlock` to commit the changes and allow other changes to be made. Any change to a shared object results in notification to all the clients that are currently subscribed to the shared object.

When you get a reference to a shared object owned by a remote application instance, any changes made to the object are sent to the instance that owns the object. The success or failure of any changes are sent using the `SharedObject.onSync` event handler, if it is defined. Also, the `SharedObject.lock` and `SharedObject.unlock` methods cannot be used to lock or unlock proxied shared objects.

## Method summary for the SharedObject object

Method	Description
<code>SharedObject.clear</code>	Deletes all properties of a persistent shared object.
<code>SharedObject.close</code>	Unsubscribes from a shared object.
<code>SharedObject.flush</code>	Causes the server to save the current state of a shared object.
<code>SharedObject.get</code>	Returns a reference to a shared object.
<code>SharedObject.getProperty</code>	Gets the value of a shared object property.
<code>SharedObject.getPropertyNames</code>	Returns an array of all the current valid properties in the shared object.
<code>SharedObject.lock</code>	Locks the shared object instance. Prevents any changes to this object by clients until the <code>SharedObject.unlock</code> method is called.
<code>SharedObject.purge</code>	Causes the server to remove all deleted properties that are older than the specified version.
<code>SharedObject.send</code>	Sends a message to the client subscribing to this shared object.
<code>SharedObject.setProperty</code>	Sets a new value for a shared object property.
<code>SharedObject.size</code>	Returns the number of valid properties in a shared object.
<code>SharedObject.unlock</code>	Unlocks a shared object instance that was locked with <code>SharedObject.lock</code> .

## Property summary for the SharedObject object

Property	Description
<code>SharedObject.name</code>	The name of a shared object.
<code>SharedObject.resyncDepth</code>	The depth that indicates when the deleted values of a shared object should be permanently deleted.
<code>SharedObject.version</code>	The current version number of a shared object.

## Event summary for the SharedObject object

Property	Description
<code>SharedObject.handlerName</code>	A placeholder for a property name that specifies a function object that is invoked when a shared object receives a broadcast message whose method name matches the property name.
<code>SharedObject.onStatus</code>	Reports errors, warnings, and status messages for a shared object.
<code>SharedObject.onSync</code>	Invoked when a shared object changes.

## SharedObject.clear

### Availability

Flash Communication Server MX.

### Usage

`SharedObject.clear()`

**Parameters**

None.

**Returns**

Returns `true` if successful; `false` otherwise.

**Description**

Method; deletes all properties and sends a “clear” event to all clients that subscribe to a persistent shared object. The persistent data object is also removed from persistent shared object. You can use `SharedObject.clear` to detach from a shared object immediately after `SharedObject.get` is invoked. You can use `SharedObject.clear` when you do not want to use a shared object anymore and want to remove it from the server completely. This method lets you create shared objects that persist only for a specified time.

**Example**

The following example calls the `clear` method on the shared object `myShared`:

```
var myShared = SharedObject.get("foo", true);  
var len = myShared.clear();
```

## SharedObject.close

**Availability**

Flash Communication Server MX.

**Usage**

```
SharedObject.close()
```

**Parameters**

None.

**Returns**

Nothing.

**Description**

Method; detaches a reference from a shared object. A call to the `SharedObject.get` method returns a reference to a shared object instance. The reference is valid until the variable that holds the reference is no longer in use and the script is garbage-collected. To destroy a reference immediately, you can call `SharedObject.close`. You can use `SharedObject.close` when you don't want to proxy a shared object any longer.

**Example**

In this example, `myS0` is attached as a reference to shared object `foo`. When you call `myS0.close` you detach the reference `myS0` from the shared object `foo`.

```
myS0 = SharedObject.get("foo");  
    // insert code here  
myS0.close();
```

**See also**

`SharedObject.get`

## SharedObject.flush

### Availability

Flash Communication Server MX.

### Usage

```
SharedObject.flush()
```

### Parameters

None.

### Returns

A Boolean value of `true` if successful, `false` otherwise.

### Description

Method; causes the server to save the current state of the shared object instance. The shared object must have been created with the persistence option.

### Example

The following example places a reference to the shared object `foo` in the variable `myShared`. It then locks the shared object instance so that no one can make any changes to it, and then saves the shared object by calling `myShared.flush`. After the shared object is saved, it is unlocked so that further changes can be made.

```
var myShared = SharedObject.get("foo", true);
myShared.lock();
// insert code here that operates on the shared object
myShared.flush();
myShared.unlock();
```

## SharedObject.get

### Availability

Flash Communication Server MX.

### Usage

```
SharedObject.get(name, persistence [, netConnection])
```

### Parameters

*name* Name of the shared object instance to return.

*persistence* A Boolean value: `true` for a persistent shared object; `false` for a nonpersistent shared object. If no value is specified, the default value is `false`.

*netConnection* A `NetConnection` object that represents a connection to an application instance. You can pass this parameter to get a reference to a shared object on another server or a shared object that is owned by another application instance. All update notifications for the shared object specified by the *name* parameter are proxied to this instance, and the remote instance notifies the local instance when a persistent shared object changes. The `NetConnection` object that is used as the *netConnection* parameter does not need to be connected when you call `SharedObject.get`. The server connects to the remote shared object when the `NetConnection` state changes to connected. This parameter is optional.

### Returns

A reference to a shared object instance.

## Description

Static method; returns a reference to a shared object instance. To perform any operation on a shared object, the server-side script must get a reference to the named shared object using the `SharedObject.get` method. If the object requested is not found, a new instance is created.

There are two types of shared objects, persistent and nonpersistent, and they are in separate name spaces. This means that a persistent and a local shared object can have the same name, but they are two distinct shared objects. Shared objects are scoped to the name space of the application instance and are identified by a string name. The shared object names should conform to the URI specification.

You can also call `SharedObject.get` to get a reference to a shared object that is in a name space of another application instance. This instance can be on the same server or on a different server and is called a *proxied shared object*. To get a reference to a shared object from another instance, create a `NetConnection` object and use the `NetConnection.connect` method to connect to the application instance that owns the shared object. Pass the `NetConnection` object as the *netConnection* parameter of the `SharedObject.get` method. The server-side script must get a reference to a proxied shared object before there is a request for the shared object from any client. To do this, call `SharedObject.get` in the `application.onAppStart` handler.

If you call `SharedObject.get` with a *netConnection* parameter and the local application instance already has a shared object with the same name, the shared object converts to a proxied shared object. All shared object messages for clients connected to a proxied shared object are sent to the master instance.

If the connection state of the `NetConnection` object that was used as the *netConnection* parameter changes state from connected to disconnected, the proxied shared object is set to idle and any messages received from subscribers are discarded. The `NetConnection.onStatus` handler is called when a connection is lost. You can then re-establish a connection to the remote instance and call `SharedObject.get`, which changes the state of the proxied shared object from idle to connected.

If you call `SharedObject.get` with a new `NetConnection` object on a proxied shared object that is already connected and if the URI of the new `NetConnection` object doesn't match the current `NetConnection` object, the proxied shared object unsubscribes from the previous shared object, sends a "clear" event to all subscribers, and subscribes to the new shared object instance. When a subscribe to a proxied shared object is successful, all subscribers are reinitialized to the new state. This process lets you migrate a shared object from one application instance to another without disconnecting the clients.

Updates received by proxied shared objects from subscribers are checked to see if the update can be rejected based on the current state of the proxied shared object version and the version of the subscriber. If the change can be rejected, the proxied shared object doesn't forward the message to the remote instance; the reject message is sent to the subscriber.

The corresponding client-side `ActionScript` method is `SharedObject.getRemote`.

### Example

This example creates a shared object named `foo` inside the function `onProcessCmd`. The function is passed a parameter, `cmd`, that is assigned to a property in the shared object.

```
function onProcessCmd(cmd){
    // insert code here
    var shObj = SharedObject.get("foo", true);
    propName = cmd.name;
    shObj.getProperty (propName, cmd.newAddress);
}
```

The following example uses a proxied shared object. A proxied shared object resides on a server or in an application instance (called *master*) that is different than the server or application instance that the client connects to (called *proxy*). When the client connects to the proxy and gets a remote shared object, the proxy connects to the master and gives the client a reference to this shared object. The following code is in the `main.asc` file:

```
application.appStart = function() {
    nc = new NetConnection();
    nc.connect("rtmp://" + master_server + "/" + master_instance);
    proxySO = SharedObject.get("myProxy",true,nc);
    // Now, whenever the client asks for a persistent
    // shared object called myProxy they will receive myProxy
    // shared object from the master_server/master_instance
};
```

## SharedObject.getProperty

### Availability

Flash Communication Server MX.

### Usage

*mySharedObject*.getProperty(*name*)

### Parameters

*name* The name of the property in the shared object.

### Returns

The value of a SharedObject property.

### Description

Method; retrieves the value of a named property in a shared object. The returned value is a copy associated with the property, and any changes made to the returned value do not update the shared object. To update a property, use the `SharedObject.setProperty` method.

### Example

The following example gets the value of the `name` property and passes it to the `value` variable:

```
value = sharedInfo.getProperty(name);
```

### See also

`SharedObject.setProperty`

## SharedObject.getPropertyNames

### Availability

Flash Communication Server MX.

### Usage

```
mySharedObject.getPropertyNames()
```

### Parameters

None.

### Returns

An array containing all the property names of a shared object.

### Description

Method; enumerates all the property names for a given shared object. This method returns an array of strings that refer to the current properties.

### Example

This example calls `getPropertyNames` on the `myInfo` shared object and places the names into the `names` variable. It then enumerates those property names in a `for` loop.

```
myInfo = SharedObject.get("foo");
var addr = myInfo.getProperty("address");
myInfo.setProperty("city", San Francisco);
var names = myInfo.getPropertyNames();
for (x in names){
    var propVal = myInfo.getProperty(names[x]);
    trace("Value of property " + names[x] + " = " + propVal);
}
```

## SharedObject.handlerName

### Availability

Flash Communication Server MX.

### Usage

```
SharedObject.onBroadcastMsg = function([p1,..., pN]){
    // insert code here
};
```

### Parameters

*p1, ..., pN* Optional parameters passed to the handler method if the message contains user-defined parameters. These parameters are the user-defined JavaScript objects passed to the `SharedObject.send` method.

### Returns

Any return value is ignored by the server.

### Description

Event handler; a placeholder for a property name (`onBroadcastMsg` in the Usage example above) that specifies a function object that is invoked when the shared object receives a broadcast message whose method name matches the property name.

The `this` keyword used in the body of the function is set to the shared object instance returned by `SharedObject.get`.

If you don't want the server to receive a particular broadcast message, do not define this handler.

#### Example

The following example defines a handler function called `broadcastMsg`:

```
var mySO = SharedObject.get("userList", false);
mySO.broadcastMsg = function(msg1, msg2){
    trace(msg1 + " : " + msg2);
};
```

## SharedObject.lock

#### Availability

Flash Communication Server MX.

#### Usage

`SharedObject.lock()`

#### Parameters

None.

#### Returns

An integer indicating the lock count: 0 or greater indicates success; -1 indicates failure. For proxied shared objects, always returns -1.

#### Description

Method; locks the shared object instance. This method gives the server-side script exclusive access to the shared object; when the `SharedObject.unlock` method is called, all changes are batched and one update message is sent to all the clients subscribed to this shared object. If you nest the `SharedObject.lock` and `SharedObject.unlock` methods, make sure there is an `unlock` for every `lock`; otherwise, clients are blocked from accessing the shared object.

You cannot use the `SharedObject.lock` method on proxied shared objects.

#### Example

This example locks the `myShared` shared object, executes the code that is to be inserted, and then unlocks the object:

```
var myShared = SharedObject.get("foo");
myShared.lock();
// insert code here that operates on the shared object
myShared.unlock();
```

## SharedObject.name

#### Availability

Flash Communication Server MX.

#### Usage

`SharedObject.name`

#### Description

Property (read-only); the name of a shared object.

### Example

This example outputs `foo` to the NetConnection Debugger:

```
myS0 = SharedObject.get("foo");  
trace(myS0.name);
```

## SharedObject.onStatus

### Availability

Flash Communication Server MX.

### Usage

```
SharedObject.onStatus = function(info) {  
    // insert code here  
};
```

### Parameters

*info* An information object. For more information, see the Appendix, “Client-Side Information Objects,” in the *Client-Side Communication ActionScript Dictionary*.

### Returns

Nothing.

### Description

Event handler; reports errors, warnings, and status messages associated with either a local instance of a shared object or a persistent shared object.

### Example

The following example defines an `onStatus` event handler for the shared object `soInstance`:

```
soInstance = SharedObject.get("foo", true);  
soInstance.onStatus = function(infoObj){  
    //Handle S0 status messages  
};
```

## SharedObject.onSync

### Availability

Flash Communication Server MX.

### Usage

```
SharedObject.onSync = function(list){  
    return;  
};
```

### Parameters

*list* An array of objects that contain information about the properties of a shared object that have changed since the last time the `onSync` method was called. The notifications for proxied shared objects are different than for shared objects that are owned by the local application instance. The following tables list the descriptions for each type of shared object.

## Local shared objects

Code	Meaning
change	A property was changed by a subscriber.
delete	A property was deleted by a subscriber.
name	The name of a property that has changed or been deleted.
oldValue	The old value of a property. This is true for both <code>change</code> and <code>delete</code> messages; on the client, <code>oldValue</code> is not set for <code>delete</code> .

**Note:** Changing or deleting a property on the server side using the `SharedObject.setProperty` method always succeeds, so there is no notification of these changes.

## Proxied shared objects

Code	Meaning
success	A server change of the shared object was accepted.
reject	A server change of the shared object was rejected. The value on the remote instance was not changed.
change	A property was changed by another subscriber.
delete	A property was deleted. This notification can occur when a server deletes a shared object or if another subscriber deletes a property.
clear	All the properties of a shared object are deleted. This can happen when the server's shared object is out of sync with the master shared object or when the persistent shared object migrates from one instance to the other. This event is typically followed by a <code>change</code> message to restore all the server's shared object properties.
name	The name of a property that has changed or been deleted.
oldValue	The old value of the property. This is only valid for the <code>reject</code> , <code>change</code> and <code>delete</code> codes.

**Note:** The `SharedObject.onSync` method is called when a shared object has been successfully synchronized with the server. The list object may be empty if there is no change in the shared object.

### Returns

Nothing.

### Description

Event handler; invoked when a shared object changes. You should set this method to point to a function you define in order to receive notification of changes made by other subscribers for shared objects owned by the local application instance. You should also set this method to get the status of changes made by the server as well as by other subscribers for proxied shared objects.

### Example

The following example creates a function that is invoked whenever a property of the shared object so changes:

```
// create a new NetConnection object
nc = new NetConnection();
nc.connect("rtmp://server1.xyx.com/myApp");
// create the shared object
so = SharedObject.get("MasterUserList", true, nc);
// the list parameter is an array of objects containing information
// about successfully of unsuccessfully changed properties
// from the last time onSync() was called
so.onSync = function(list) {
    for (var i = 0; i < list.length; i++) {
        switch (list[i].code ) {
            case "success":
                trace ("success");
                break;
            case "change":
                trace ("change");
                break;
            case "reject":
                trace ("reject");
                break;
            case "delete":
                trace ("delete");
                break;
            case "clear":
                trace ("clear");
                break;
        }
    }
};
```

## SharedObject.purge

### Availability

Flash Communication Server MX.

### Usage

`SharedObject.purge(version)`

### Parameters

*version* A version number. All deleted data that is older than this version is removed.

### Returns

Nothing.

### Description

Method; causes the server to purge all deleted properties that are older than the specified version. Although you can also accomplish this task by setting the `SharedObject.resyncDepth` property, the `SharedObject.purge` method gives the script more control over which properties to delete.

### Example

This example deletes all properties of the `myShared` shared object that are older than the value of `SharedObject.version - 3`.

```
var myShared = SharedObject.get("foo", true);
myShared.lock();
myShared.purge(myShared.version - 3);
myShared.unlock();
```

## SharedObject.resyncDepth

### Availability

Flash Communication Server MX.

### Usage

`SharedObject.resyncDepth`

### Description

Property; an integer that indicates when the deleted properties of a shared object should be permanently deleted. You can use this property in a server-side script to resynchronize shared objects and to control when shared objects are deleted. The default value is infinity.

If the current revision number of the shared object minus the revision number of the deleted property is greater than the value of `SharedObject.resyncDepth`, the property is deleted. Also, if a client connecting to this shared object has a client revision that, when added to the value of `SharedObject.resyncDepth` is less than the value of the current revision on the server, all the current elements of the client shared object are deleted, the valid properties are sent to the client and the client receives a “clear” message.

This method is useful when you add and delete many properties and you don't want to send too many messages to the Flash client. Suppose a client is connected to a shared object that has 12 properties and then disconnects. After that client disconnects, other clients that are connected to the shared object delete 20 properties and add 10 properties. When the client reconnects, it could, for example, receive a delete message for the 10 properties it previously had and then a change message on two properties. You could use `SharedObject.resyncDepth` to send a “clear” message, followed by a change message for two properties, which saves the client from receiving 10 delete messages.

### Example

The following example resynchronizes the shared object `mySO` if the revision number difference is greater than 10:

```
mySo = SharedObject.get("foo");
mySo.resyncDepth = 10;
```

## SharedObject.send

### Availability

Flash Communication Server MX.

### Usage

`SharedObject.send(methodName, [p1, ..., pN])`

### Parameters

*methodName* The name of a method on a client shared object instance. For example, if you specify `doSomething`, the client must invoke the `SharedObject.doSomething` method, with all the *p1*, ..., *pN* parameters.

*p1*, ..., *pN* Parameters of any `ActionScript` type, including references to other `ActionScript` objects. These parameters are passed to the specified *methodName* when it executes on the Flash client.

### Returns

A Boolean value of `true` if the message was sent to the client; `false` otherwise.

### Description

Method; executes a method in a client-side script. You can use `SharedObject.send` to asynchronously execute a method on all the Flash clients subscribing to a shared object. The server does not receive any notification from the client on the success, failure, or return value in response to this message.

### Example

This example calls the `SharedObject.send` method to execute the `doSomething` method in the client-side `ActionScript` and passes `doSomething` the string `"this is a test"`.

```
var myShared = SharedObject.get("foo", true);
myShared.send("doSomething", "this is a test");
```

The following example is the client-side `ActionScript` code that defines the `doSomething` method:

```
connection = new NetConnection();
connection.connect("rtmp://www.macromedia.com/someApp");
var x = SharedObject.getRemote("foo", connection.uri, true);
x.connect(connection);
x.doSomething = function(str) {
    // process the str
};
```

## SharedObject.setProperty

### Availability

Flash Communication Server MX.

### Usage

```
sharedInfo.setProperty(name, value);
```

### Parameters

*name* The name of the property in the shared object.

*value* An `ActionScript` object associated with the property, or `null` to delete the property.

### Returns

Nothing.

### Description

Method; updates the value of a property in a shared object. A shared object property can be modified by another user of the shared object between successive calls to `SharedObject.getProperty` and `SharedObject.setProperty`. If you want to preserve transactional integrity, call the `SharedObject.lock` method before operating on the shared object; be sure to call `SharedObject.unlock` when the operations finish. If you don't call `SharedObject.lock` and the `SharedObject.setProperty` is called, the change is made to the shared object, and all subscribers of the object are notified before `SharedObject.setProperty` returns. If a call to the `SharedObject.lock` method precedes a call to `SharedObject.setProperty`, all changes are batched and sent when the `SharedObject.unlock` method is called.

If you call `SharedObject.setProperty` on the server side, it invokes a change message in the `SharedObject.onSync` method on the client side for any Flash Player client that is using the shared object. The *name* parameter on the server side is the same as an attribute of the data property on the client side. For example, the following two lines of code are equivalent: the first line is server-side ActionScript, and the second is client-side ActionScript:

```
sharedInfo.setProperty(nameVal, "foo");
clientSO.data[nameVal] = "foo";
```

### Example

This example uses the `SharedObject.setProperty` method to create the property `city` with the value `San Francisco`. It then enumerates all the property values in a `for` loop and prints out the results in the Output window by using a `trace` action.

```
myInfo = SharedObject.get("foo");
var addr = myInfo.getProperty("address");
myInfo.setProperty("city", "San Francisco");
var names = sharedInfo.getPropertyNames();
for (x in names){
    var propVal = sharedInfo.getProperty(names[x]);
    trace("Value of property " + names[x] + " = " + propVal);
}
```

### See also

`SharedObject.getProperty`

## SharedObject.size

### Availability

Flash Communication Server MX.

### Usage

```
mySharedObject.size()
```

### Parameters

None.

### Returns

An integer indicating the number of properties.

### Description

Method; returns the total number of valid properties in a shared object.

### Example

This example gets the number of properties of a shared object and assigns that number to the variable `len`:

```
var myShared = SharedObject.get(application, "foo", true);  
var len = myShared.size();
```

## SharedObject.unlock

### Availability

Flash Communication Server MX.

### Usage

```
mySharedObject.unlock()
```

### Parameters

None.

### Returns

An integer indicating the lock count: 0 or greater if successful, -1 otherwise. For proxied shared objects, this method always returns -1.

### Description

Method; unlocks a shared object instance. This causes the script to relinquish exclusive access to the shared object and lets other clients update the instance. This method also causes the server to commit all changes made after the `SharedObject.lock` method is called and sends an update message to all clients.

You cannot use the `SharedObject.unlock` method on proxied shared objects.

### Example

This example illustrates how easy it is to unlock a shared object.

```
var myShared = SharedObject.get("foo", true);  
myShared.lock();  
// insert code to manipulate the shared object  
myShared.unlock();
```

### See also

`SharedObject.lock`

## SharedObject.version

### Availability

Flash Communication Server MX.

### Usage

```
SharedObject.version
```

### Description

Property (read-only); the current version number of the shared object. Changes made to the shared object either by the Flash Player client or by the server-side script using the `SharedObject.setProperty` method increment the value of the version property.

## Stream (object)

The Stream object lets you handle each stream in a Flash Communication Server application. A *stream* is a one-way connection between the Flash Player and the Flash Communication Server, or between two servers. The Flash Communication Server automatically creates a Stream object with a unique name when `NetStream.play` or `NetStream.publish` are called in a client-side script. You can also create a stream in server-side ActionScript by calling `Stream.get`. A user can access multiple streams at the same time, and there can be hundreds or thousands of Stream objects active at the same time.

You can use the properties and methods of the Stream object to shuffle streams in a playlist, pull streams from other servers, and play and record streams.

You can also use the Stream object to play MP3 files over a stream, as well as ID3 tags associated with MP3 files.

You can create other Stream properties of any legal ActionScript type, including references to other ActionScript objects, for a particular instance of the Stream object. The properties are available until the stream is removed from the application.

For more information about streams, see the NetStream (object) entry in the *Client-Side Communication ActionScript Dictionary*.

### Property summary for the Stream object

Property (read-only)	Description
<code>Stream.bufferTime</code>	Indicates how long to buffer messages before a stream is played.
<code>Stream.name</code>	The unique name of a live stream.

### Method summary for the Stream object

Method	Description
<code>Stream.clear</code>	Deletes a stream previously recorded by the server.
<code>Stream.get</code>	Returns a reference to a Stream object. This is a static method.
<code>Stream.length</code>	Returns the length of a recorded stream in seconds. This is a static method.
<code>Stream.play</code>	Controls the data source of the Stream object.
<code>Stream.record</code>	Records all the data going into the stream.
<code>Stream.send</code>	Sends a call with parameters to all subscribers on a stream.
<code>Stream.setBufferTime</code>	Sets the length of the buffer time in seconds.

### Event handler summary for the Stream object

Event handler	Description
<code>Stream.onStatus</code>	Called when there is a change in status.

## Stream.bufferTime

### Availability

Flash Communication Server MX.

### Usage

`Stream.bufferTime`

### Description

Property (read-only); indicates how long to buffer messages before a stream is played. This property applies only when playing a stream from a remote server or when playing a recorded stream locally. To set this property use `Stream.setBuffertime`.

A message is data that is sent back and forth between the Flash Communication Server and the Flash Player. The data is divided into small packets (messages), and each message has a type (audio, video, or data).

### See also

`Stream.setBufferTime`

## Stream.clear

### Availability

Flash Communication Server MX.

### Usage

`Stream.clear()`

### Parameters

None.

### Returns

A Boolean value of `true` if the call succeeds, `false` otherwise.

### Description

Method; deletes a recorded stream file (FLV) from the server.

### Example

This example deletes a recorded stream called `foo.flv`. Before the stream is deleted, the example defines an `onStatus` handler that uses two information object error codes, `NetStream.Clear.Success` and `NetStream.Clear.Failed`, to send status messages to the Output window.

```
s = Stream.get("foo");
if (s){
    s.onStatus = function(info){
        if(info.code == "NetStream.Clear.Success"){
            trace("Stream cleared successfully.");
        }
        if(info.code == "NetStream.Clear.Failed"){
            trace("Failed to clear stream.");
        }
    };
    s.clear();
}
```

**Note:** The Stream information object is nearly identical to the client-side ActionScript NetStream information object. For more information, see the Appendix, "Server-Side Information Objects," on page 77.

**See also**

Stream.get

## Stream.get

**Availability**

Flash Communication Server MX.

**Usage**

Stream.get(*name*)

**Parameters**

*name* The name of the stream instance to return.

**Returns**

A reference to a stream instance.

**Description**

Method (static); returns a reference to a Stream object. If the requested object is not found, a new instance is created.

**Examples**

This example gets the stream `foo` and assigns it to the variable `playStream`. It then calls the `Stream.play` method from `playStream`.

```
function onProcessCmd(cmd){
    var playStream = Stream.get("foo");
    playStream.play("file1", 0, -1);
}
```

**See also**

Stream.clear

## Stream.length

**Availability**

Flash Communication Server MX.

**Usage**

Stream.length(*name*)

**Parameters**

*name* Name of a recorded stream (FLV) file or MP3 file. To get the length of an MP3 file, precede the name of the file with `mp3:` (for example, `"mp3:beethoven"`).

**Returns**

The length of a recorded stream file or MP3 file in seconds.

**Description**

Method (static); returns the length of a recorded stream file or MP3 file in seconds. If the requested file is not found, the return value is 0.

### Example

This example gets the length of the recorded stream file `myVideo` and assigns it to the variable `streamLen`:

```
function onProcessCmd(cmd){
    var streamLen = Stream.length("myVideo");
    trace("Length: " + streamLen + "\n");
}
```

This example gets the length of the MP3 file `beethoven.mp3` and assigns it to the variable `streamLen`:

```
function onProcessCmd(cmd){
    var streamLen = Stream.length("mp3:beethoven");
    trace("Length: " + streamLen + "\n");
}
```

## Stream.name

### Availability

Flash Communication Server MX.

### Usage

`Stream.name`

### Description

Property (read-only); contains a unique string associated with a live stream.

You can also use this as an index to find a stream within an application.

### Example

The following function, `getStreamName`, takes a stream reference as an argument, `myStream`, and returns the name of the stream associated with it.

```
function getStreamName (myStream) {
    return myStream.name;
}
```

## Stream.onStatus

### Availability

Flash Communication Server MX.

### Usage

```
myStream.onStatus = function([infoObject]) {
    // Insert code here
};
```

### Parameters

*infoObject* An optional parameter defined according to the status message. For details about this parameter, see Appendix, “Server-Side Information Objects,” on page 77.

### Returns

Nothing.

### Description

Event handler; invoked every time the status of a Stream object changes. For example, if you play a file in a stream, `Stream.onStatus` is invoked. Use `Stream.onStatus` to check when play starts and ends, when recording starts, and so on.

### Example

This example defines a function that executes whenever the `Stream.onStatus` event is invoked:

```
s = Stream.get("foo");
s.onStatus = function(info){
    // insert code here
};
```

## Stream.play

### Availability

Flash Communication Server MX.

### Usage

```
Stream.play(streamName, [startTime, length, reset, remoteConnection])
```

### Parameters

*streamName* The name of any published live stream, recorded stream (FLV file), or MP3 file.

To play back FLV files, the default Flash file format for recorded streams, specify the name of the stream (for example, "myVideo"). To play back MP3 files that you've stored on the server, or the ID3 tags of MP3 files, you must precede the stream name with `mp3:` or `id3:` (for example, "mp3:bolero" or "id3:bolero"). For more information on playing MP3 files, see *Developing Communication Applications*.

*startTime* The start time of the stream playback, in seconds. If no value is specified, it is set to -2. If *startTime* is -2, the server tries to play a live stream with the name specified in *streamName*. If no live stream is available, the server tries to play a recorded stream with the name specified in *streamName*. If no recorded stream is found, the server creates a live stream with the name specified in *streamName* and waits for someone to publish to that stream. If *startTime* is -1, the server attempts to play a live stream with the name specified in *streamName* and waits for a publisher if no specified live stream is available. If *startTime* is greater than or equal to 0, the server plays the recorded stream with the name specified in *streamName*, starting from the time given. If no recorded stream is found, the `play` method is ignored. If a negative value other than -1 is specified, the server interprets it as -2. This is an optional parameter.

*length* The length of play, in seconds. For a live stream, a value of -1 plays the stream as long as the stream exists. Any positive value plays the stream for the corresponding number of seconds. For a recorded stream, a value of -1 plays the entire file, and a value of 0 returns the first video frame. Any positive number plays the stream for the corresponding number of seconds. By default, the value is -1. This is an optional parameter.

*reset* A Boolean value, or number, that flushes the playing stream. If *reset* is `false` (0), the server maintains a playlist, and each call to `Stream.play` is appended to the end of the playlist so that the next play does not start until the previous play finishes. You can use this technique to create a dynamic playlist. If *reset* is `true` (1), any playing stream stops, and the playlist is reset. By default, the value is `true`.

You can also specify a number value of 2 or 3 for the *reset* parameter, which is useful when playing recorded stream files that contain message data. These values are analogous to *false* (0) and *true* (1), respectively: a value of 2 maintains a playlist, and a value of 3 resets the playlist. However, the difference is that specifying either 2 or 3 for *reset* returns all messages in the specified recorded stream at once, rather than at the intervals which the messages were originally recorded (the default behavior).

For more information on Flash Communication Server logging see Flash Communication Server TechNote 16464 on the Macromedia Flash Communication Server Support Center, [www.macromedia.com/support/flashcom/ts/documents/flashcom\\_logging.htm](http://www.macromedia.com/support/flashcom/ts/documents/flashcom_logging.htm).

*remoteConnection* A reference to a NetConnection object that is used to connect to a remote server. The requested stream plays from the remote server if this parameter is provided. This parameter is optional.

### Returns

A Boolean value: *true* if the *Stream.play* call is accepted by the server and added to the playlist; *false* otherwise. The *Stream.play* method can fail if the server fails to find the stream or if an error occurs. To get information about the *Stream.play* call, you can define a *Stream.onStatus* handler to catch the play status or error.

If the *streamName* parameter is *false*, the stream stops playing. A Boolean value of *true* is returned if the stop succeeds; *false* otherwise.

### Description

Method; controls the data source of a stream with an optional start time, duration, and reset flag to flush any previously playing stream. The *Stream.play* method also has a parameter that lets you reference a NetConnection object to play a stream from another server. The *Stream.play* method allows you to do the following:

- Chain streams between servers.
- Create a hub to switch between live streams and recorded streams.
- Combine different streams into a recorded stream.

You can combine multiple streams to create a playlist for clients. The server-side *Stream.play* method behaves a bit differently than the *NetStream.play* method on the client side. A *play* call on the server is similar to a *publish* call on the client. It controls the source of the data that is coming into a stream. When you call *Stream.play* on the server, the server becomes the publisher. Because the server has a higher priority than the client, the client is forced to unpublish from the stream if the server calls a *play* method on the same stream.

In general, if any recorded streams are included in a server playlist, you cannot play the server stream as a live stream.

**Note:** A stream that plays from a remote server by means of the NetConnection object is a live stream.

If you require a value to begin a stream, you may need to change the Application.xml file's "Enhanced seeking" flag at the server. "Enhanced seeking" is a Boolean flag in the Application.xml file. By default, this flag is set to *false*. When a play occurs, the server seeks to the closest video keyframe possible and starts from that keyframe. For example, if you want to play at time 15, and there are keyframes only at time 11 and time 17, seeking will start from time 17 instead of time 15. This is an approximate seeking method that works well with compressed streams.

If the flag is set to `true`, some compression is invoked on the server. Using the previous example, if the flag is set to `true`, the server creates a keyframe—based on the preexisting keyframe at time 11—for each keyframe from 11 through 15. Even though a keyframe does not exist at the seek time, the server generates a keyframe, which involves some processing time on the server.

### Example

This example illustrates how streams can be chained between servers:

```
application.myRemoteConn = new NetConnection();
application.myRemoteConn.onStatus = function(info){
    trace("Connection to remote server status " + info.code + "\n");
    // tell all the clients
    for (var i = 0; i < application.clients.length; i++){
        application.clients[i].call("onServerStatus", null,
            info.code, info.description);
    }
};
// Use the NetConnection object to connect to a remote server
application.myRemoteConn.connect(rtmp://movie.com/movieApp);
// Setup the server stream
application.myStream = Stream.get("foo");
if (application.myStream){
    application.myStream.play("Movie1", 0, -1, true, application.myRemoteConn);
}
```

The following example shows `Stream.play` used as a hub to switch between live streams and recorded streams:

```
// Set up the server stream
application.myStream = Stream.get("foo");
if (application.myStream){
    // this server stream will play "Live1",
    // "Record1", and "Live2" for 5 seconds each
    application.myStream.play("Live1", -1, 5);
    application.myStream.play("Record1", 0, 5, false);
    application.myStream.play("Live2", -1, 5, false);
}
```

The following example combines different streams into a recorded stream:

```
// Set up the server stream
application.myStream = Stream.get("foo");
if (application.myStream){
    // Like the previous example, this server stream
    // will play "Live1", "Record1", and "Live2"
    // for 5 seconds each. But this time,
    // all the data will be recorded to a recorded stream "foo".
    application.myStream.record();
    application.myStream.play("Live1", -1, 5);
    application.myStream.play("Record1", 0, 5, false);
    application.myStream.play("Live2", -1, 5, false);
}
```

The following example uses `Stream.play` to stop playing the stream `foo`:

```
application.myStream.play(false);
```

The following example creates a playlist of three MP3 files (beethoven.mp3, mozart.mp3, and chopin.mp3) and plays each file in turn over the live stream foo:

```
application.myStream = Stream.get("foo");
if(application.myStream) {
    application.myStream.play("mp3:beethoven", 0);
    application.myStream.play("mp3:mozart", 0, false);
    application.myStream.play("mp3:chopin.mp3", 0, false);
}
```

In the following example, data messages in the recorded stream file log.flv are returned at the intervals which they were originally recorded.

```
application.myStream = Stream.get("data");
if (application.myStream) {
    application.myStream.play("log", 0, -1);
}
```

In the following example, data messages in the recorded stream file log.flv are returned all at once, rather than at the intervals which they were originally recorded.

```
application.myStream = Stream.get("data");
if (application.myStream) {
    application.myStream.play("log", 0, -1, 2);
}
```

## Stream.record

### Availability

Flash Communication Server MX.

### Usage

`Stream.record(flag)`

### Parameters

*flag* This parameter can have the value `record`, `append`, or `false`. If the value is `record`, the data file is overwritten if it exists. If the value is `append`, the incoming data is appended to the end of the existing file. If the value is `false`, any previous recording stops. By default, the value is `record`.

### Returns

A Boolean value of `true` if the recording succeeds, `false` otherwise.

### Description

Method; records all the data going through a Stream object.

### Example

This example opens a stream `s` and, when it is open, plays `sample` and records it. Because no value is passed to the `record` method, the default value, `record`, is passed.

```
// To start recording
s = Stream.get("foo");
if (s){
    s.play("sample");
    s.record();
}
// To stop recording
s = Stream.get("foo");
if (s){
    s.record(false);
}
```

## Stream.send

### Availability

Flash Communication Server MX.

### Usage

`Stream.send(handlerName, [p1, ..., pN])`

### Parameters

*handlerName* Calls the specified handler in client-side ActionScript code. The *handlerName* value is the name of a method relative to the subscribing Stream object. For example, if *handlerName* is `doSomething`, the `doSomething` method at the stream level is invoked with all the *p1, ..., pN* parameters. Unlike the method names in `Client.call` and `NetConnection.call`, the handler name can be only one level deep (that is, it cannot be of the form *object/method*).

**Note:** Do not use a built-in method name for a handler name. For example, the subscribing stream will be closed if the handler name is `close`.

*p1, ..., pN* Parameters of any ActionScript type, including references to other ActionScript objects. These parameters are passed to the specified handler when it is executed on the Flash client.

### Returns

A Boolean value of `true` if the message was sent to the client, `false` otherwise.

### Description

Method; sends a message to all clients subscribing to the stream and the message is processed by the handler specified on the client. Because the server has higher priority than the clients, the server can still send a message on a stream owned by a client. Unlike the `Stream.play()` method, the server does not need to take ownership of a stream from the client in order to send a message. After `send()` is called, the client still owns the stream as a publisher.

### Example

This example calls the method `Test` on the client-side Stream object and sends it the string "hello world":

```
application.streams["foo"].send("Test", "hello world");
```

The following example is the client-side ActionScript that receives the `Stream.send` call. The method `Test` is defined on the `Stream` object:

```
tStream.Test = function(str) {  
    // insert code to process the str  
}
```

## Stream.setBufferTime

### Availability

Flash Communication Server MX.

### Usage

```
Stream.setBufferTime()
```

### Description

Method; increases the message queue length. When you play a stream from a remote server, the `Stream.setBufferTime` method sends a message to the remote server that adjusts the length of the message queue. The default length of the message queue is 2 seconds. You should set the buffer time higher when playing a high-quality recorded stream over a low-bandwidth network.

When a user clicks a seek button in an application, buffered packets are sent to the server. The buffered seeking in a Flash Communication Server application occurs on the server; the Flash Communication Server doesn't support client-side buffering. The seek time can be smaller or larger than the buffer size, and it has no direct relationship to the buffer size. Every time the server receives a seek request from the Flash Player, it clears the message queue on the server. The server tries to seek to the desired position and starts filling the queue again. At the same time, the Flash Player also clears its own buffer after a seek request, and the buffer is eventually filled after the server starts sending the new messages.

# trace

## Availability

Flash Communication Server MX.

## Usage

```
trace("Hello world");  
trace("Value of i = " + i);
```

## Returns

Nothing.

## Description

Method (global); displays the value of an expression in the Output window. The `trace` message is also published to the logs/application appname stream, which is available in the Administration Console or in the Communication App inspector. The values in the `trace` expression are converted to strings if possible before they are sent to the Output window. You can use the `trace` method to debug a script.



# APPENDIX

## Server-Side Information Objects

The Application, NetConnection, and Stream objects provide an `onStatus` event handler that uses an information object for providing information, status, or error messages. To respond to this event handler, you must create a function to process the information object, and you must know the format and contents of the information object returned.

You can define the following global function at the top of your `main.asc` file to display all the status messages for the parameters that you pass to the function. You need to place this code in the `main.asc` file only once.

```
function showStatusForClass(){
    for (var i=0;i<arguments.length;i++){
        arguments[i].prototype.onStatus = function(infoObj){
            trace(infoObj.code + " (level:" + infoObj.level + ")");
        }
    }
}
showStatusForClass(NetConnection, Stream);
```

For more information about information objects, see the appendix of the *Client-Side Communication ActionScript Dictionary*.

An information object has the following properties: `level`, `code`, `description`, and `details`. All information objects have `level` and `code` properties, but only some have the `description` and/or `details` properties. The following tables list the `code` and `level` properties as well as the meaning of each information object.

### Application information objects

The following table lists the information objects for the Application object.

Code	Level	Meaning
<code>Application.Script.Error</code>	Error	The ActionScript engine has encountered a runtime error. In addition to the standard <i>infoObject</i> properties, the following properties are set: <code>filename</code> : name of the offending ASC file. <code>lineno</code> : line number where the error occurred. <code>linebuf</code> : source code of the offending line.
<code>Application.Script.Warning</code>	Warning	The ActionScript engine has encountered a runtime warning. In addition to the standard <i>infoObject</i> properties, the following properties are set: <code>filename</code> : name of the offending ASC file. <code>lineno</code> : line number where the error occurred. <code>linebuf</code> : source code of the offending line.

Code	Level	Meaning
<code>Application.Resource.LowMemory</code>	Warning	The ActionScript engine is low on runtime memory. This provides an opportunity for the application instance to free some resources or take suitable action. If the application instance runs out of memory, it is unloaded and all users are disconnected. In this state, the server will not invoke the <code>Application.onDisconnect</code> event handler or the <code>Application.onAppStop</code> event handler.
<code>Application.Shutdown</code>	Status	This information object is passed to the <code>onAppStop</code> handler when the application is being shut down.
<code>Application.GC</code>	Status	This information object is passed to the <code>onAppStop</code> event handler when the application instance is about to be destroyed by the server.

## NetConnection information objects

The `NetConnection` object has the same information objects as the client-side `NetConnection` object.

Code	Level	Meaning
<code>NetConnection.Call.Failed</code>	Error	The <code>NetConnection.call</code> method was not able to invoke the server-side method or command.*
<code>NetConnection.Connect.AppShutdown</code>	Error	The application has been shut down (for example, if the application is out of memory resources and must shut down to prevent the server from crashing) or the server has shut down.
<code>NetConnection.call.BadVersion</code>	Error	The URI specified in the <code>NetConnection.connect</code> method did not specify “rtmp” as the protocol. “rtmp” must be specified when connecting to Flash Communication Server.
<code>NetConnection.Connect.Closed</code>	Status	The connection was closed successfully.
<code>NetConnection.Connect.Failed</code>	Error	The connection attempt failed.
<code>NetConnection.Connect.InvalidApp</code>	Error	The application name specified during the connection attempt was not found on the server.
<code>NetConnection.Connect.Rejected</code>	Error	The client does not have permission to connect to the application, or the application expected different parameters from those that were passed.**
<code>NetConnection.Connect.Success</code>	Status	The connection attempt succeeded.

\* This information object also has a `description` property, which is a string that provides a specific reason for the failure.

\*\* This information object also has an `application` property, which contains the value that the `application.rejectConnection` server-side method returns.

## Stream information objects

The information objects of the Stream object are similar to those of the client-side NetStream object.

Code	Level	Meaning
NetStream.Clear.Success	Status	A recorded stream was deleted successfully.
NetStream.Clear.Failed	Error	A recorded stream failed to delete.
NetStream.Publish.Start	Status	An attempt to publish was successful.
NetStream.Publish.BadName	Error	An attempt was made to publish a stream that is already being published by someone else.
NetStream.Failed	Error	An attempt to use a Stream method failed.*
NetStream.Unpublish.Success	Status	An attempt to unpublish was successful.
NetStream.Record.Start	Status	Recording was started.
NetStream.Record.NoAccess	Error	An attempt was made to record a read-only stream.
NetStream.Record.Stop	Status	Recording was stopped.
NetStream.Record.Failed	Error	An attempt to record a stream failed.
NetStream.Play.Start	Status	Play was started.**
NetStream.Play.StreamNotFound	Error	An attempt was made to play a stream that does not exist.
NetStream.Play.Stop	Status	Play was stopped.
NetStream.Play.Failed	Error	An attempt to play back a stream failed.* **
NetStream.Play.Reset	Status	A playlist was reset.
NetStream.Play.PublishNotify	Status	The initial publish to a stream was successful. This message is sent to all subscribers.
NetStream.Play.UnpublishNotify	Status	An unpublish from a stream was successful. This message is sent to all subscribers.

\* This information object also has a `description` property, which is a string that provides a specific reason for the failure.

\*\* This information object also has a `details` property, which is a string that provides the name of the streams being played. This is useful for multiple plays. The `details` property shows the name of the stream when switching from one element in the playlist to the next element.

